



GPT-5.4

Нейроперевод

РАЗРАБОТКА ОТВЕТСТВЕННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Практическое руководство по авиационному ПО и стандарту DO-178C

ЛИАННА РИЕРСОН



Разработка ответственного программного обеспечения

Практическое руководство по авиационному программному обеспечению и соответствию DO-178C (KT-178C). LEANNA RIERSON

Посетите сайт Taylor & Francis: <http://www.taylorandfrancis.com>

Я посвящаю эту книгу памяти Cary Spitzer, который верил в ее важность и в мою способность ее написать, а также моей бабушке Charlotte Richardson, которая ежедневно молилась о моей работе. И Cary, и бабушка Richardson ушли из жизни осенью 2011 года, когда я завершала первый черновик этой книги. Мне их обоих очень не хватает, и я надеюсь, что они были бы довольны текстом, на создание которого они меня вдохновили.

Содержание

Разработка ответственного программного обеспечения	
Содержание	1
Предисловие	15
Благодарности	16
Автор	18
Часть I. Введение	19
Глава 1. Введение и обзор	19
Сокращения	19
1.1 Определение программного обеспечения, критичного по безопасности	19
1.2 Важность акцента на безопасности	20
1.3 Назначение книги и важные оговорки	22
1.4 Обзор книги	25
Литература	26
Часть II. Контекст разработки программного обеспечения, критичного по безопасности	27
Глава 2. Программное обеспечение в контексте системы	28
Сокращения	28
2.1 Обзор системной разработки	28
2.2 Системные требования	32
2.2.1 Важность системных требований	32
2.2.2 Типы системных требований	32
2.2.3 Характеристики хороших требований	33
2.2.4 Аспекты системных требований, которые следует учитывать	35
2.2.5 Допущения в требованиях	39
2.2.6 Распределение по элементам	40
2.3 Валидация и верификация системных требований	40
2.3.1 Валидация требований	40
2.3.2 Верификация реализации	40
2.3.3 Рекомендации по валидации и верификации	41
2.4 Лучшие практики для системных инженеров	44
2.5 Связь программного обеспечения с системой	47
Литература	49
Глава 3. Программное обеспечение в контексте оценки безопасности системы	51
Сокращения	51

3.1 Обзор процесса оценки безопасности воздушного судна и системы	51
3.1.1 План программы обеспечения безопасности	53
3.1.2 Оценка функциональной опасности	53
3.1.3 Оценка функциональных опасностей системы	55
3.1.4 Предварительная оценка безопасности воздушного судна	55
3.1.5 Предварительная оценка безопасности системы	55
3.1.6 Анализ общих причин	56
3.1.7 Оценка безопасности воздушного судна и системы	58
3.2 Гарантия разработки	58
3.2.1 Уровни гарантии разработки	60
3.3 Как программное обеспечение вписывается в процесс обеспечения безопасно- сти?	62
3.3.1 Уникальность программного обеспечения	62
3.3.2 Гарантия разработки программного обеспечения	63
3.3.3 Другие точки зрения	64
3.3.4 Некоторые предложения по учету программного обеспечения в системном процессе обеспечения безопасности	65
Литература	66
Часть III. Разработка программного обеспечения, критичного по безопасности, с использованием DO-178C (KT-178C)	68
Глава 4. Обзор DO-178C и сопутствующих документов	69
Сокращения	69
4.1 История DO-178	71
4.2 Основные документы DO-178C и DO-278A	75
4.2.1 Отличия DO-278A и DO-178C	78
4.2.2 Обзор таблиц целей DO-178C из Annex A	79
4.3 DO-330: Соображения по квалификации программных инструментов	83
4.4 Технологические дополнения к DO-178C	84
4.4.1 DO-331: Дополнение по модельно-ориентированной разработке	84
4.4.2 DO-332: Дополнение по объектно-ориентированной технологии	85
4.4.3 DO-333: Дополнение по формальным методам	86
4.5 DO-248C: Вспомогательный материал	86
Литература	87
Глава 5. Планирование программного обеспечения	89
Сокращения	89
5.1 Введение	90

5.2 Общие рекомендации по планированию	90
5.3 Пять программных планов	93
5.3.1 План сертификации ПО (PSAC)	94
5.3.2 План разработки ПО (Software Development Plan, SDP)	96
5.3.3 План верификации ПО (Software Verification Plan, SVP)	99
5.3.4 План управления конфигурацией ПО (Software Configuration Management Plan, SCMP)	102
5.3.5 План гарантии качества ПО (Software Quality Assurance Plan, SQAP)	107
5.4 Три стандарта разработки	109
5.4.1 Стандарты требований к программному обеспечению	110
5.4.2 Стандарты проектирования программного обеспечения	111
5.4.3 Стандарты кодирования программного обеспечения	112
5.5 Планирование квалификации инструментов	113
5.6 Другие планы	114
5.6.1 План управления проектом	114
5.6.2 План управления требованиями	114
5.6.3 План тестов	114
Литература	114
Глава 6. Требования к программному обеспечению	116
Сокращения	116
6.1 Введение	116
6.2 Определение требования	117
6.3 Важность хороших требований к программному обеспечению	118
6.3.1 Причина 1: Требования являются фундаментом разработки программного обеспечения	118
6.3.2 Причина 2: Хорошие требования экономят время и деньги	121
6.3.3 Причина 3: Хорошие требования необходимы для безопасности	121
6.3.4 Причина 4: Хорошие требования необходимы для удовлетворения потреб- ностей заказчика	122
6.3.5 Причина 5: Хорошие требования важны для тестирования	122
6.4 Инженер по требованиям к программному обеспечению	123
6.5 Обзор разработки требований к программному обеспечению	125
6.6 Сбор и анализ входных данных для требований к программному обеспечению	127
6.6.1 Мероприятия по сбору требований	127
6.6.2 Мероприятия по анализу требований	128
6.7 Написание требований к программному обеспечению	129

6.7.1 Задача 1: Определить методологию	129
6.7.2 Задача 2: Определить компоновку документа требований высокого уровня к ПО (Software Requirements Document, SWRD)	131
6.7.3 Задача 3: Разделить функциональность программного обеспечения на под- системы и (или) функции	132
6.7.4 Задача 4: Определить приоритеты требований	133
6.7.5 Краткое отступление (не задача): опасные уклоны, которых следует избегать	133
6.7.6 Задача 5: Задokumentировать требования	136
6.7.7 Задача 6: Предоставить обратную связь по системным требованиям	144
6.8 Верификация (рассмотрение) требований	144
6.8.1 Рекомендуемые практики экспертного рассмотрения	147
6.9 Управление требованиями	150
6.9.1 Основы управления требованиями	150
6.9.2 Инструменты управления требованиями	152
6.10 Прототипирование требований	154
6.11 Трассируемость	155
6.11.1 Важность и преимущества трассируемости	155
6.11.2 Двухнаправленная трассируемость	156
6.11.3 DO-178C и трассируемость	158
6.11.4 Трудности трассируемости	159
Литература	162
Рекомендуемое чтение	163
Глава 7. Проектирование программного обеспечения	165
Сокращения	165
7.1 Обзор проектирования программного обеспечения	165
7.1.1 Архитектура программного обеспечения	165
7.1.2 Требования низкого уровня к ПО	166
7.1.3 Компоновка проектной документации	169
7.2 Подходы к проектированию	170
7.2.1 Структурное проектирование (традиционный подход)	170
7.2.2 Объектно-ориентированное проектирование	173
7.3 Характеристики хорошего проекта	174
7.4 Верификация проекта	181
Литература	183
Глава 8. Реализация программного обеспечения: кодирование и интеграция . . .	184
Сокращения	184

8.1 Введение	185
8.2 Кодирование	185
8.2.1 Обзор указаний DO-178C по кодированию	185
8.2.2 Языки, используемые в критичном по безопасности программном обеспе- чении	186
8.2.3 Выбор языка и компилятора	190
8.2.4 Общие рекомендации по кодированию	192
8.2.5 Специальные темы, связанные с кодом	206
8.3 Верификация исходного кода	207
8.4 Интеграция в ходе разработки	208
8.4.1 Процесс сборки	209
8.4.2 Процесс загрузки	211
8.5 Верификация интеграции в ходе разработки	211
Список литературы	211
Рекомендуемая литература	212
Глава 9. Верификация программного обеспечения	214
Сокращения	214
9.1 Введение	215
9.2 Важность верификации	216
9.3 Независимость и верификация	217
9.4 Рассмотрения	218
9.4.1 Рассмотрение планирования программного обеспечения	219
9.4.2 Рассмотрения требований, проекта и кода	219
9.4.3 Рассмотрения тестовых данных	219
9.4.4 Рассмотрение других элементов данных	219
9.5 Анализы	220
9.5.1 Анализ наихудшего времени выполнения	221
9.5.2 Анализ запасов памяти	222
9.5.3 Анализ компоновки и карты памяти	223
9.5.4 Анализ загрузки	223
9.5.5 Анализ прерываний	223
9.5.6 Математический анализ	224
9.5.7 Анализ ошибок и предупреждений	225
9.5.8 Анализ обособления	225
9.6 Тестирование программного обеспечения	225
9.6.1 Назначение тестирования программного обеспечения	226

9.6.2 Обзор указаний DO-178C по тестированию программного обеспечения . .	228
9.6.3 Обзор стратегий тестирования	230
9.6.4 Планирование тестирования	236
9.6.5 Разработка тестов	238
9.6.6 Выполнение тестирования	242
9.6.7 Отчетность по тестированию	244
9.6.8 Трассируемость тестирования	245
9.6.9 Регрессионное тестирование	245
9.6.10 Пригодность к тестированию	247
9.6.11 Автоматизация в процессах верификации	247
9.7 Верификация верификации	248
9.7.1 Рассмотрение тестовых процедур	249
9.7.2 Рассмотрение результатов тестирования	251
9.7.3 Анализ покрытия требований	251
9.7.4 Анализ структурного покрытия	252
9.8 Сообщения о проблемах	262
9.9 Рекомендации по процессам верификации	267
Ссылки	272
Рекомендуемое чтение	273
Глава 10. Управление конфигурацией ПО	277
Сокращения	277
10.1 Введение	277
10.1.1 Что такое управление конфигурацией ПО?	277
10.1.2 Зачем нужно управление конфигурацией ПО?	278
10.1.3 Кто отвечает за внедрение управления конфигурацией ПО?	280
10.1.4 Что включает в себя управление конфигурацией ПО?	281
10.2 Мероприятия по управлению конфигурацией ПО	282
10.2.1 Идентификация конфигурации	282
10.2.2 Базовые версии	283
10.2.3 Трассируемость	283
10.2.4 Сообщения о проблемах	284
10.2.5 Управление изменениями и их рассмотрение	289
10.2.6 Учет состояния конфигурации	291
10.2.7 Выпуск	292
10.2.8 Архивирование и извлечение	293
10.2.9 Категории управления данными	294

10.2.10 Управление загрузкой	296
10.2.11 Управление средой жизненного цикла ПО	296
10.3 Специальные навыки в управлении конфигурацией ПО	298
10.4 Данные управления конфигурацией ПО	298
10.4.1 План управления конфигурацией ПО (SCMP)	298
10.4.2 Сообщения о проблемах	299
10.4.3 Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index, SECI)	299
10.4.4 Указатель конфигурации ПО (Software Configuration Index, SCI)	299
10.4.5 Записи управления конфигурацией ПО	301
10.5 Подводные камни управления конфигурацией ПО	301
10.6 Анализ воздействия изменений	304
Ссылки	309
Глава 11. Гарантия качества ПО	311
Сокращения	311
11.1 Введение: качество ПО и гарантия качества ПО (Software Quality Assurance, SQA)	311
11.1.2 Характеристики высококачественного программного обеспечения	312
11.1.3 Гарантия качества ПО	313
11.1.4 Примеры распространенных проблем качества процесса и продукта	316
11.2 Характеристики эффективной и неэффективной гарантии качества ПО	316
11.2.1 Эффективная гарантия качества ПО	316
11.2.2 Неэффективная гарантия качества ПО	318
11.3 Мероприятия по гарантии качества ПО	319
Ссылки	324
Глава 12. Взаимодействие по вопросам сертификации	327
Сокращения	327
12.1 Что такое взаимодействие с сертифицирующим органом?	328
12.2 Взаимодействие с сертифицирующими органами	329
12.2.1 Рекомендуемые практики координации с сертифицирующими органами	330
12.3 Итоговый отчёт разработки ПО	334
12.4 Аудиты этапов взаимодействия с сертифицирующим органом	336
12.4.1 Обзор аудитов этапов взаимодействия	336
12.4.2 Обзор документа Software Review Job Aid	337
12.4.3 Использование пособия Software Job Aid	344
12.4.4 Общие рекомендации для аудитора	345

<i>История о характерах #1: неприятные телеконы</i>	348
<i>История о характерах #2: я боялась за свою жизнь</i>	349
12.4.5 Общие рекомендации для аудируемой стороны (заявителя/разработчика)	352
12.4.6 Особенности аудитов этапов взаимодействия	355
12.5 Зрелость программного обеспечения перед сертификационными летными испытаниями	367
Ссылки	368
Часть IV. Квалификация инструментов и дополнения к DO-178C (КТ-178C)	371
Глава 13. DO-330 и квалификация программных инструментов	371
Сокращения	371
13.1 Введение	372
13.2 Определение необходимости и уровня квалификации инструмента (раздел 12.2 DO-178C)	375
13.3 Квалификация инструмента (обзор DO-330)	379
13.3.1 Необходимость DO-330	380
13.3.2 Процесс квалификации инструмента по DO-330	381
13.4 Специальные темы квалификации инструмента	395
13.4.1 Приказ FAA 8110.49	395
13.4.2 Детерминизм инструмента	396
13.4.3 Дополнительные соображения по квалификации инструмента	397
13.4.4 Подводные камни квалификации инструмента	399
13.4.5 DO-330 и приложения к DO-178C	401
13.4.6 Использование DO-330 в других доменах	402
Ссылки	402
Глава 14. DO-331 и разработка и верификация на основе моделей	403
Сокращения	403
14.1 Введение	403
14.2 Потенциальные преимущества разработки и верификации на основе моделей	405
14.3 Потенциальные риски разработки и верификации на основе моделей	409
14.4 Обзор DO-331	413
14.5 Признание DO-331 сертифицирующими органами	422
Ссылки	423
Глава 15. DO-332 и объектно-ориентированная технология и связанные с ней методы	425
Сокращения	425
15.1 Введение в объектно-ориентированную технологию	425
15.2 Использование объектно-ориентированной технологии в авиации	427

15.3 Справочник по объектно-ориентированной технологии в авиации	427
15.4 Исследования объектно-ориентированной технологии и структурного покрытия, финансируемые Федеральным управлением гражданской авиации США	428
15.5 Обзор DO-332	429
15.5.1 Планирование	430
15.5.2 Разработка	430
15.5.3 Верификация	430
15.5.4 Уязвимости	431
15.5.5 Безопасность типов	431
15.5.6 Связанные методы	432
15.5.7 Часто задаваемые вопросы	432
15.6 Рекомендации по объектно-ориентированной технологии	433
15.7 Заключение	434
Ссылки	434
Рекомендуемое чтение[*]	435
Глава 16. DO-333 и формальные методы	436
Сокращения	436
16.1 Введение в формальные методы	436
16.2 Что такое формальные методы?	438
16.3 Потенциальные преимущества формальных методов	441
16.4 Трудности формальных методов	442
16.5 Обзор DO-333	444
16.5.1 Назначение DO-333	444
16.5.2 Сравнение DO-333 и DO-178C	444
16.6 Другие ресурсы	448
Ссылки	448
Часть V Специальные темы	450
Глава 17. Непокрытый код (мертвый, посторонний и отключенный код)	450
Сокращения	450
17.1 Введение	451
17.2 Посторонний и мертвый код	451
17.2.1 Как избежать позднего обнаружения постороннего и мертвого кода	453
17.2.2 Оценка постороннего или мертвого кода	454
17.3 Отключенный код	457
17.3.1 Планирование	461
17.3.2 Разработка	462

17.3.3 Верификация	463
Ссылки	464
Глава 18. Загружаемое в полевых условиях программное обеспечение	465
Сокращения	465
18.1 Введение	465
18.2 Что такое загружаемое в полевых условиях программное обеспечение? . . .	466
18.3 Преимущества загружаемого в полевых условиях программного обеспечения	467
18.4 Трудности загружаемого в полевых условиях программного обеспечения . .	467
18.5 Разработка и загрузка загружаемого в полевых условиях программного обес- печения	468
18.5.1 Разработка системы, пригодной для полевой загрузки	468
18.5.2 Разработка загружаемого в полевых условиях программного обеспечения	469
18.5.3 Загрузка загружаемого в полевых условиях программного обеспечения .	470
18.5.4 Модификация загружаемого в полевых условиях программного обеспечения	472
18.6 Заключение	472
Ссылки	472
Глава 19. Модифицируемое пользователем ПО	473
Сокращения	473
19.1 Введение	473
19.2 Что такое модифицируемое пользователем ПО?	473
19.3 Примеры модифицируемого пользователем ПО	475
19.4 Проектирование системы для модифицируемого пользователем ПО	476
19.5 Модификация и сопровождение модифицируемого пользователем ПО	479
Ссылки	481
Глава 20. Операционные системы реального времени	483
Сокращения	483
20.1 Введение	484
20.2 Что такое ОСРВ?	484
Х. М. Дейтел пишет:	484
20.3 Зачем использовать ОСРВ?	485
20.4 Ядро ОСРВ и поддерживающее его программное обеспечение	487
20.4.1 Ядро ОСРВ	488
20.4.2 Интерфейс прикладного программирования	488
20.4.3 Пакет поддержки платы	490
20.4.4 Драйвер устройства	491
20.4.5 Поддерживающие библиотеки	491

20.5 Характеристики ОСРВ, используемой в системах, критичных по безопасности	491
20.5.1 Детерминированность	491
20.5.2 Надежная производительность	492
20.5.3 Совместимость с аппаратным обеспечением	492
20.5.4 Совместимость со средой	492
20.5.5 Устойчивость к неисправностям	492
20.5.6 Контроль состояния	493
20.5.7 Пригодность к сертификации	494
20.5.8 Сопровождаемость	494
20.5.9 Повторное использование	494
20.6 Функции ОСРВ, используемой в системах, критичных по безопасности . . .	494
20.6.1 Многозадачность	495
20.6.2 Гарантированная и детерминированная планируемость	495
20.6.3 Детерминированное взаимодействие между задачами	497
20.6.4 Надежное управление памятью	498
20.6.5 Обработка прерываний	498
20.6.6 Функции-перехватчики	499
20.6.7 Проверка робастности	500
20.6.8 Файловая система	500
20.6.9 Надежное обособление	500
20.7 Вопросы, которые следует учитывать при работе с ОСРВ	501
20.7.1 Технические вопросы, которые следует учитывать	501
20.7.2 Вопросы сертификации, которые следует учитывать	504
20.8 Другие темы, связанные с ОСРВ	508
20.8.1 Обзор ARINC 653	508
20.8.2 Инструментальная поддержка	511
20.8.3 Операционные системы реального времени с открытым исходным кодом	512
20.8.4 Многоядерные процессоры, виртуализация и гипервизоры	513
20.8.5 Защищенность	514
20.8.6 Вопросы выбора операционной системы реального времени	514
Ссылки	515
Глава 21. Обособление ПО	519
Сокращения	519
21.1 Введение в обособление	519
21.1.1 Обособление как подмножество защиты	519
21.1.2 DO-178C и обособление	521

21.1.3 Надежное обособление	522
21.2 Общая память (пространственное обособление)	524
21.3 Общий центральный процессор (временное обособление)	526
21.4 Общий ввод-вывод	527
21.5 Некоторые сложности, связанные с обособлением	528
21.5.1 Прямой доступ к памяти	528
21.5.2 Кэш-память	529
21.5.3 Прерывания	530
21.5.4 Межраздельное взаимодействие	530
21.6 Рекомендации по обособлению	531
Список литературы	537
Глава 22. Конфигурационные данные	539
Сокращения	539
22.1 Введение	539
22.2 Терминология и примеры	540
22.3 Сводка указаний DO-178C по параметрическим данным	543
22.4 Рекомендации	544
Ссылки	550
Глава 23. Аэронавигационные данные	551
Сокращения	551
23.1 Введение	551
23.2 DO-200A: Standards for Processing Aeronautical Data	553
23.3 Консультативный циркуляр FAA 20-153A	557
23.4 Инструменты, используемые для обработки аэронавигационных данных . .	560
23.5 Другие отраслевые документы, относящиеся к аэронавигационным данным	560
23.5.1 DO-201A: Standards for Aeronautical Information	560
23.5.2 DO-236B: Minimum Aviation System Performance Standards: Required Navigation Performance for Area Navigation	561
23.5.3 DO-272C: User Requirements for Aerodrome Mapping Information	561
23.5.4 DO-276A: User Requirements for Terrain and Obstacle Data	561
23.5.5 DO-291B: Interchange Standards for Terrain, Obstacle, and Aerodrome Mapping Data	561
23.5.6 ARINC 424: Standard, Navigation System Database	562
23.5.7 ARINC 816-1: Embedded Interchange Format for Airport Mapping Database	563
Ссылки	563
Глава 24. Повторное использование программного обеспечения	565

Сокращения	565
24.1 Введение	565
24.2 Проектирование повторно используемых компонентов	569
24.3 Повторное использование ранее разработанного ПО	575
24.3.1 Оценка ранее разработанного ПО для использования в изделиях граждан- ской авиации	575
24.3.2 Повторное использование ранее разработанного ПО, не разработанного по DO-178[]	581
24.3.3 Дополнительные соображения по готовому коммерческому ПО	582
24.4 История эксплуатации изделия	585
24.4.1 Определение истории эксплуатации изделия	585
24.4.2 Трудности при попытке получить зачет на основе истории эксплуатации изделия	586
24.4.3 Факторы, которые следует учитывать при заявлении зачета на основе ис- тории эксплуатации изделия	586
Глава 25. Обратная разработка	589
Сокращения	589
25.1 Что такое обратная разработка?	590
25.2 Примеры обратной разработки	591
25.3 Проблемы, которые необходимо учитывать при обратной разработке	591
25.4 Рекомендации по обратной разработке	593
Литература	602
Глава 26. Аутсорсинг и офшоринг мероприятий ЖЦ ПО	603
Сокращения	603
26.1 Введение	603
26.2 Причины аутсорсинга	605
26.3 Трудности и риски аутсорсинга	606
26.4 Рекомендации по преодолению трудностей и рисков	611
26.5 Резюме	622
Литература	622
Приложение А. Пример критериев перехода	624
Сокращения	624
Приложение В. Проблемные области ОСРВ	633
Сокращения	633
Литература	636

Приложение С. Вопросы, которые следует рассмотреть при выборе операционной системы реального времени (real-time operating system, RTOS; ОСРВ) для системы, критичной по безопасности	637
С.1 Общие вопросы	637
С.2 Вопросы по функциональности ОСРВ	638
С.3 Вопросы по интеграции ОСРВ	640
Литература	641
Приложение D. Вопросы по истории эксплуатации ПО	642
D.1 Вопросы, касающиеся сообщений о проблемах [1]	642
D.2 Вопросы, касающиеся эксплуатации [1]	645
D.3 Вопросы, касающиеся среды [1]	645
D.4 Вопросы, касающиеся времени [1]	646
Литература	647

Предисловие

Я представляю эту книгу с большим чувством ответственности и полной скромностью. После двух лет исследований и написания мне кажется, что я лишь слегка затронула тему. Надеюсь, что то, чем я делюсь и что испытала на собственном опыте, поможет вам в вашей профессиональной работе.

Моя страсть, это безопасность и практическая реализуемость ее достижения. Еще в старших классах школы я вела колонку в местной газете с советами по безопасности, подробно исследуя, насколько это было возможно в сельской Америке до появления интернета, безопасность в ванной, безопасность при работе на тракторе, электробезопасность и другие темы. В выпускном классе школы я участвовала в конкурсе агитационных выступлений на уровне штата, и темой была безопасность ремней безопасности. Желание продвигать безопасность привело меня в Федеральное управление гражданской авиации США (FAA) и стало главным мотивом для этой книги.

Программное обеспечение, это лишь одно звено в общей цепи безопасности. Но это важное звено, и с каждым годом оно становится все более значимым. По мере того как ПО распространяется шире и используется все в более критичных задачах, растут и связанные с ним риски, и его вклад в безопасность. Одновременно с ростом риска, связанного с ПО, бизнес-модели многих компаний, по-видимому, меняются: от людей ожидают большего при меньших ресурсах, график становится главным, а инженеров-программистов рассматривают как фигуры на игровой доске, которые можно произвольно переставлять.

Эта книга задумана как инструмент для тех, кто оказался захвачен круговоротом отрасли. Возможно, вы системный инженер или менеджер, менеджер по программному обеспечению, инженер-программист, инженер по обеспечению качества или студент, стремящийся учиться на опыте других. Вы хотите выполнять работу на выдающемся уровне, но вас подавляют сроки и бюджетные ограничения. Я надеюсь, что эта книга, основанная на последних двадцати годах моей жизни в авиационной отрасли, поможет вам в стремлении к качеству и совершенству.

Благодарности

Эта книга была бы невозможна без помощи многих коллег и друзей. Следующие люди рассмотрели отдельные части текста и предоставили ценные замечания: Пэтти Бат (Patty Bath, ранее Bartels) из AVISTA, Джим Челини (Jim Chelini) из VEROCEL, Эндрю Эллиотт (Andrew Elliott) из Design Assurance, Келли Хейхерст (Kelly Hayhurst) из исследовательского центра NASA в Лэнгли, Венди Льюнгрен (Wendy Ljungren) из GE Aviation, Джефф Никербокер (Jeff Knickerbocker) из Sunrise Certification, Джордж Романски (George Romanski) из VEROCEL, Эндрю Корнецки (Andrew Kornecki) из авиационного университета Embry-Riddle, Фред Стратманн (Fred Strathmann) из Springton Technologies, Стивен Вандерлист (Steven VanderLeest) из DornerWorks и Дэн Вирхузен (Dan Veerhusen), вышедший на пенсию из Rockwell Collins. Благодарю каждого из вас за ваше время и дружбу. Вы заставляли меня переосмысливать свои взгляды и сделали текст лучше.

Эта книга вообще не могла бы появиться без тех, кто наставлял меня, направлял и открывал передо мной двери возможностей в начале моей карьеры. Их слишком много, чтобы перечислить всех, но особенно я обязана следующим:

- Милт Силлс, бывший вице-президент по инженерным вопросам компании Cessna Aircraft Company. Он был моим наставником и разжег мою любовь к авиации. Он ушел из жизни в 2003 году, но продолжает ежедневно влиять на мою жизнь.
- Джим Питерсон, бывший менеджер FAA по системам и оборудованию в Wichita Aircraft Certification Office. Джим привлек меня в FAA и привил мне стремление отстаивать правильное, какой бы ни была цена.
- Элизабет Эрикссон, бывший директор FAA по сертификации авиационной техники, а ныне сотрудник Erickson Aviation Services. Бет верила в меня и наставляла меня в трудные времена в штаб-квартире FAA в Вашингтоне, округ Колумбия.
- Шерил Дорси, владелица Digital Flight. Шерил ввела меня в ряды инженеров по программному обеспечению в штаб-квартире FAA. Мой опыт был связан с авиационными системами и электронной аппаратурой, но Шерил увидела мой потенциал и пробудила во мне стремление к безопасному ПО.
- Майк ДеУолт, главный научно-технический советник FAA по авиационному компьютерному ПО. Майк направлял меня, когда я была очень молода и неопытна.

Я также благодарна RTCA, Inc. Мне выпала честь работать в пяти комитетах, и в трех из них, в руководящих ролях. RTCA оказала этой книге большую поддержку и любезно дала разрешение на ссылки и цитаты. Больше узнать о RTCA, Inc. или приобрести их документы можно, связавшись с ними по адресу:

RTCA, Inc. 1150 18th Street NW Suite 910 Washington, DC 20036 Телефон: (202) 833-9339

Факс: (202) 833-9434 Веб-сайт: www.rtca.org

Также благодарю моих дорогих друзей Сьюзан Коркоран (Susan Corcoran) и Дениз Полански (Denise Polansky), которые помогали на заключительных этапах этой работы, команду, которая молилась обо мне до самого финиша, а также сотрудников Taylor & Francis Group за их поддержку и помощь. Нора Конопка (Nora Konopka), Джилл Юргенсен (Jill Jurgensen) и Гленон Батлер (Glenon Butler) были великолепны.

MATLAB[®] и Simulink[®] являются зарегистрированными товарными знаками The MathWorks, Inc. За информацией о продукте обращайтесь:

The MathWorks, Inc. 3 Apple Hill Drive Natick, MA, 01760-2098 USA Тел.: 508-647-7000

Факс: 508-647-7001 E-mail: info@mathworks.com Веб-сайт: www.mathworks.com



Лиэнна Рирсон является независимым консультантом по программному обеспечению, сложной электронной аппаратуре и разработке интегрированной модульной авионики (ИМА) для систем, критичных по безопасности, с акцентом на гражданскую авиацию. У нее более 20 лет опыта в области ПО и авиации. Рирсон девять лет работала специалистом по ПО и авионике в Федеральном управлении гражданской авиации США (FAA), причем пять из этих лет, в должности главного научно-технического советника по авиационному компьютерному ПО. Рирсон опубликовала множество работ по ПО, критичному по безопасности, интегрированной модульной авионике и авиации, руководила многими национальными и международными инженерными группами и семинарами, а также разрабатывала для FAA учебные курсы, политики, справочные руководства и методические материалы. Она была сопредседателем подгруппы и руководителем редакционной группы в специальном комитете RTCA, который подготовил DO-178C и еще шесть связанных с ним документов. Рирсон преподавала DO-178B, а теперь и DO-178C, сотням специалистов. Она является уполномоченным инженерным представителем FAA (DER) с полномочиями уровня А в технических областях ПО и сложной аппаратуры. Она работала со многими авиационными и авионическими компаниями, включая Boeing, Cessna, Learjet, Embraer, Rockwell Collins, GE Aviation, Honeywell и многие другие. В настоящее время она работает неполный рабочий день в группе сертификации авионики Rockwell Collins.

Примечание переводчика. Далее по тексту основными обозначениями оставлены DO-178 и DO-254; российские аналоги КТ-178 и КТ-254 приводятся точно там, где это уместно как справка.

Рирсон имеет степень магистра по программной инженерии Рочестерского технологического института и степень бакалавра по электротехнике Университета штата Уичито. Она также училась в Ozark Christian College и получила степень магистра в Johnson Bible College. В настоящее время она продолжает обучение в докторантуре.

Часть I. Введение

Часть I закладывает основу для всей книги. В ней объясняется, что такое программное обеспечение, критичное по безопасности, и почему оно важно в современных условиях. Также дается обзор книги и объясняется принятый подход.

Глава 1. Введение и обзор

Сокращения

Сокращение	Расшифровка
DER	Designated Engineering Representative (назначенный инженерный представитель)
EASA	European Aviation Safety Agency (Европейское агентство по авиационной безопасности)
FAA	Federal Aviation Administration (Федеральное управление гражданской авиации США)
IEEE	Institute of Electrical and Electronic Engineers (Институт инженеров по электротехнике и электронике)
IMA	Integrated Modular Avionics (интегрированная модульная авионика)
NASA	National Aeronautics and Space Administration (Национальное управление по аэронавтике и исследованию космического пространства)

1.1 Определение программного обеспечения, критичного по безопасности

Общее определение *безопасности* таково: это “свобода от тех условий, которые могут вызвать смерть, травму, болезнь, повреждение или утрату оборудования или имущества либо вред окружающей среде” [1]. Определение *программного обеспечения, критичного по безопасности* более субъективно. Институт инженеров по электротехнике и электронике (IEEE) определяет ПО, критичное по безопасности, следующим образом: “ПО, использование которого в системе может привести к неприемлемому риску. ПО, критичное по безопасности, включает ПО, работа или отказ которого может привести к опасному состоянию, ПО, предназначенное для восстановления после опасных состояний, и ПО, предназначенное для смягчения тяжести аварии” [2]. *Стандарт безопасности ПО*, опубликованный Национальным управлением по аэронавтике и исследованию космического пространства США (NASA), относит ПО к *критичному по безопасности*, если выполняется хотя бы один из следующих критериев [3,4]:

1. Оно находится в системе, критичной по безопасности, как это определено анализом опасностей, И при этом выполняется хотя бы одно из следующего:
 - Вызывает опасность или способствует ее возникновению.
 - Обеспечивает управление опасностями или их смягчение. Управляет функциями, критичными по безопасности.
 - Обрабатывает команды или данные, критичные по безопасности.
 - Обнаруживает и сообщает либо предпринимает корректирующее действие, если система достигает определенного опасного состояния.
 - Смягчает ущерб, если опасность реализуется.
 - Находится в той же системе, то есть на том же процессоре, что и программное обеспечение, критичное по безопасности.
2. Оно обрабатывает данные или анализирует тенденции, которые напрямую ведут к решениям, связанным с безопасностью.
3. Оно обеспечивает полную или частичную верификацию или валидацию систем, критичных по безопасности, включая аппаратные или программные системы.

Из этих определений можно заключить, что программное обеспечение само по себе не является ни безопасным, ни небезопасным; однако, когда оно является частью системы, критичной по безопасности, оно может вызывать небезопасные условия или способствовать им. Такое ПО считается *критичным по безопасности* и является главной темой этой книги.

1.2 Важность акцента на безопасности

В 1993 году Рут Винер (Ruth Wiener) написала в своей книге *Digital Woes: Why We Should Not Depend Upon Software*: “Программные продукты, даже программы скромного размера, относятся к самым сложным артефактам, которые создают люди, а проекты по разработке программного обеспечения – к числу самых сложных наших начинаний. Они поглощают столько времени или денег, сколько им дать, и столько людей, сколько на них бросить. Результаты лишь умеренно надежны. Даже после самого тщательного и строгого тестирования некоторые ошибки остаются. Мы никогда не сможем протестировать все пути прохождения через систему при всех возможных входных данных” [5]. С тех пор общество стало все больше и больше зависеть от ПО. Мы уже прошли точку, в которой можно было бы вернуться к чисто аналоговым системам. Следовательно, мы должны прилагать все усилия к тому, чтобы системы, в значительной степени зависящие от ПО, были надежными и безопасными. У авиационной отрасли хороший послужной список, но по мере роста сложности и критичности необходимо проявлять осторожность.

Нередко можно услышать высказывания вроде: “Программное обеспечение не вызывало

крупных авиационных катастроф, так из-за чего весь шум?” Вклад ПО в авиационные происшествия является предметом споров, поскольку ПО является частью более крупной системы, а большинство расследований сосредоточено на аспектах уровня системы. Однако в других областях, например в ядерной, медицинской и космической, ошибки ПО определенно приводили к гибели людей или потере миссии. Взрыв ракеты Ariane Five, передозировки облучения аппаратом Therac-25 и отказ ракетной системы Patriot во время войны в Персидском заливе являются одними из наиболее известных примеров. Историческая статистика по ПО, критичному по безопасности, в гражданской авиации выглядит весьма достойно. Однако сейчас не время откинуться на спинку кресла и восхищаться прошлым. Будущее рискованнее по следующим причинам:

- *Увеличение числа строк кода:* количество строк кода, используемого в системах, критичных по безопасности, растет. Например, объем кода от Boeing 777 к недавно сертифицированному Boeing 787 увеличился в восемь-десять раз, а в будущих самолетах программного обеспечения будет еще больше.
- *Рост сложности:* сложность систем и ПО возрастает. Например, интегрированная модульная авионика (IMA) обеспечивает уменьшение массы, более простую установку, эффективное обслуживание и снижение стоимости изменений. Однако IMA также повышает сложность системы и приводит к тому, что функции, ранее изолированные на физически различном федеративном оборудовании, объединяются на единой аппаратной платформе, с сопутствующей потерей нескольких функций при отказе этого оборудования. Этот рост сложности затрудняет как тщательный анализ влияния на безопасность, так и доказательство того, что требуемая функциональность выполняется без непреднамеренных последствий.
- *Рост критичности:* одновременно с увеличением размера и сложности ПО возрастает и его критичность. Например, еще 10 лет назад интерфейсы рулевых поверхностей почти исключительно были механическими. Теперь многие производители самолетов переходят к системе fly-by-wire (электродистанционная система управления), в которой рулевые поверхности управляются ПО, поскольку такое ПО включает алгоритмы, улучшающие летные характеристики и устойчивость самолета.
- *Изменения технологий:* технологии электроники и ПО меняются чрезвычайно быстро. Сложно обеспечить зрелость технологии до того, как она устареет. Например, в критичных по безопасности областях требуются надежные и проверенные микропроцессоры; однако, поскольку разработка нового самолета занимает около 5 лет, микропроцессоры нередко оказываются почти устаревшими уже к моменту летных испытаний изделия. Кроме того, изменения в технологиях ПО затрудняют найм программистов, знающих Assembly, C или Ada, то есть наиболее распространенные языки, используемые для

бортового ПО. Эти языки реального времени во многих университетах не преподаются. Разработчики ПО также все дальше уходят от фактически генерируемого машинного кода.

- *Больше меньшими силами:* из-за экономических факторов и давления на прибыль многие, а возможно и большинство, инженерных организаций получают указание делать больше меньшими силами. Я слышала, как это описывали так: “Сначала у нас забрали секретарей; потом забрали технических писателей; а затем забрали коллег”. Многие хорошие инженеры делают ту работу, которую раньше выполняли два и более человека. Они перегружены и истощены. После многих месяцев переработок люди становятся менее эффективными. Один мой коллега формулирует это так: “Сверхурочная работа годится для спринтов, а не для марафонов”. И все же многие инженеры работают сверхурочно месяцами, а то и годами.
- *Рост аутсорсинга и офшоринга:* из-за требований рынка и нехватки инженеров все больше ПО, критичного по безопасности, передается на аутсорсинг и в офшор. Это не всегда проблема, но зачастую у аутсорсинговых и офшорных команд нет ни знаний системной предметной области, ни необходимого опыта в области безопасности, чтобы эффективно находить и устранять критические ошибки. Более того, без надлежащего надзора они могут даже вносить ошибки.
- *Убыль опытных инженеров:* многие инженеры, благодаря которым и был достигнут нынешний уровень безопасности, уходят на пенсию. Без строгой программы обучения и наставничества более молодые инженеры и руководители не понимают, почему были приняты ключевые решения и введены ключевые практики. Поэтому они либо не следуют им, либо вовсе добиваются их отмены. Недавно один мой коллега выразил свое изумление после того, как его команда сделала для нового системного инженера их сертифицирующего органа, выступающего системным *специалистом*, презентацию по воздушным тормозам. После двухчасовой презентации новый инженер спросил: “Вы ведь не о колесах говорите?”[*]
- *Недостаток доступных программ обучения:* существует очень мало образовательных программ с акцентом на безопасность. Кроме того, формального образования в области валидации и верификации систем и ПО крайне мало.

Из-за этих и других факторов риска необходимо уделять безопасности еще больше внимания, чем когда-либо прежде.

1.3 Назначение книги и важные оговорки

Для меня большая честь, и я по-настоящему смиренно отношусь к тому, что вы решили прочитать эту книгу. Цель этого текста – снабдить практикующих инженеров и руководителей информацией, необходимой им для разработки программного обеспечения, критич-

ного по безопасности, для авиации. За последние 20 лет мне выпала привилегия работать разработчиком авионики, разработчиком воздушного судна, представителем сертифицирующего органа, назначенным инженерным представителем (DER) Федерального управления гражданской авиации США (FAA), консультантом и преподавателем. Я оценивала ПО, критичное по безопасности, на десятках систем, таких как системы управления полетом, ИМА, шасси, закрылки, противообледенительная защита, управление передней стойкой шасси, управление полетом, управление аккумуляторами, индикация, навигация, предупреждение о рельефе и столкновении с землей, предупреждение о столкновении с воздушными судами и предотвращение столкновений, ОСРВ (RTOS) и многие другие. Я работала и с командами по 500 человек, и с командами по 3 человека. Это было и остается захватывающим приключением.

Разнообразие должностей и систем позволило мне увидеть и типичные проблемы, и эффективные решения при разработке программного обеспечения, критичного по безопасности. Эта книга написана с опорой на такой опыт, чтобы помочь практикующим системным инженерам воздушных судов, инженерам по авионике и электронным системам, менеджерам по ПО, инженерам по разработке и верификации ПО, представителям сертифицирующих органов и их уполномоченным лицам, инженерам по обеспечению качества и другим людям, стремящимся внедрять и обеспечивать безопасное ПО.

Как практическое руководство по разработке и верификации программного обеспечения, критичного по безопасности, этот текст дает конкретные указания и рекомендации, основанные на реальных проектах. Однако важно отметить следующее:

- Приведенная здесь информация представляет собой взгляд одного человека. Она написана на основе личного опыта и наблюдений, а также исследований и взаимодействия с некоторыми из самых ярких и преданных своему делу людей в мире. Были приложены все усилия, чтобы представить точные и полные рекомендации; однако каждый день я узнаю что-то новое, что уточняет и расширяет мое понимание. На протяжении всей книги используются выражения вроде *обычно*, *как правило*, *нормально*, *в большинстве случаев*, *во многих случаях*, *зачастую* и т.д. Эти обобщения сделаны на основе многочисленных проектов, в которых я участвовала; однако существует множество проектов и миллиарды строк кода, которых я не видела. Ваш опыт может отличаться, и я приветствую такой взгляд. Если вы хотите уточнить или обсудить тему, задать вопрос или поделиться своими мыслями, вы можете написать по обоим следующим адресам электронной почты: LRierson1@aol.com и Digital_Safety@sbcglobal.net.
- Я использовала личный и в некоторой степени неформальный тон. Поскольку я преподавала DO-178B, а теперь и DO-178C (KT-178C), сотням слушателей, эта книга задумана как взаимодействие между мной, преподавателем, и вами, инженером. Поскольку я

не знаю вашего опыта, я постаралась писать так, чтобы текст был полезен независимо от вашего уровня подготовки. Я включила *боевые истории* и иногда немного юмора, как делаю это в аудитории. При этом постоянной целью оставался профессионализм.

- Поскольку эта книга сосредоточена на программном обеспечении, критичном по безопасности, текст написан с ориентацией на ПО более высокой критичности, например уровней А и В по DO-178С. Для более низких уровней критичности некоторые мероприятия могут не требоваться.
- Хотя эта книга сосредоточена на авиационном ПО, соответствии DO-178С и сертификации воздушных судов, многие концепции и лучшие практики применимы и к другим областям, критичным по безопасности или по выполнению миссии, таким как медицина, ядерная энергетика, военная техника, автомобильная промышленность и космос.
- Из-за моего опыта работы в качестве представителя сертифицирующего органа и DER чтение этой книги временами может немного напоминать попытку забраться в сознание представителя сертифицирующего органа. Не бойтесь. Обсуждаются как материалы политики и руководства FAA, так и Европейского агентства по авиационной безопасности (EASA); однако, если нет существенного различия, в основном используется руководство FAA. Хотя содержание этой книги предназначено для объяснения политики и руководящих материалов сертифицирующих органов и согласуется с ними в том виде, в каком они существовали на момент написания, эта книга не является политикой или руководством сертифицирующего органа. Пожалуйста, консультируйтесь с вашим местным сертифицирующим органом по поводу политики и руководства, применимых к вашему конкретному проекту.
- Побывав в самых разных странах мира и взаимодействуя с инженерами на шести континентах, я приложила все усилия, чтобы представить международный взгляд на ПО, критичное по безопасности, и сертификацию. Однако, поскольку большая часть моей работы проходила в Соединенных Штатах и на проектах, сертифицируемых FAA, именно этот взгляд и представлен. На протяжении всего документа приводятся ссылки на несколько документов RTCA, включая DO-178С. Для меня было честью работать в руководящих ролях в трех комитетах RTCA и играть ключевую роль в разработке упоминаемых документов. Как отмечено в предисловии, RTCA любезно позволила мне ссылаться на части своих документов и цитировать их. Однако эта книга не заменяет эти документы. Я постаралась охватить основные положения и провести вас через некоторые тонкости DO-178С и связанных документов. Однако, если вы разрабатываете ПО, которое должно соответствовать документам RTCA, обязательно прочитайте документ полностью. Подробнее узнать о RTCA, Inc. или приобрести их документы можно, связавшись с ними по следующим координатам:

RTCA, Inc.

1150 18th Street NW Suite 910

Washington, DC 20036 Phone: (202) 833-9339 Fax: (202) 833-9434 Web: www.rtca.org

- Эта книга написана так, чтобы ее можно было читать от начала до конца, а можно было читать отдельные главы по мере необходимости. В некоторых главах вы обнаружите местами повторения. Это сделано намеренно, поскольку некоторые читатели могут использовать эту книгу как справочник, а не читать ее подряд. Ссылки на связанные главы приводятся по всему тексту, чтобы помочь тем, кто не будет читать книгу от корки до корки.

1.4 Обзор книги

Эта книга разделена на пять частей. Часть I, то есть эта часть, дает введение и закладывает основу. Часть II объясняет роль программного обеспечения в общей системе и содержит краткое изложение процессов системной разработки и оценки безопасности, применяемых в авиации. Часть III начинается с обзора документа RTCA DO-178C под названием *Software Considerations in Airborne Systems and Equipment Certification* и шести других документов, опубликованных вместе с DO-178C. Затем в этом разделе рассматриваются процессы DO-178C, даются пояснения к соответствующим руководящим указаниям и предложения по их эффективному применению. Часть IV рассматривает четыре руководящих документа RTCA, выпущенных вместе с DO-178C. Рассматриваемые темы таковы: квалификация программных инструментов (DO-330), разработка и верификация на основе моделей (DO-331), объектно-ориентированная технология (OOT) и связанные с ней методы (DO-332), а также формальные методы (DO-333). Часть V охватывает специальные темы, относящиеся к DO-178C и разработке ПО, критичного по безопасности. Эти специальные темы ориентированы на авиацию, но могут быть применимы и к другим областям и включают непокрытый код, то есть посторонний, мертвый и отключенный код, загружаемое в полевых условиях ПО, модифицируемое пользователем ПО (UMS), ОСРВ (RTOS), обособление, конфигурационные данные, аэронавигационные базы данных, повторное использование ПО, ранее разработанное ПО (PDS), обратную разработку, а также аутсорсинг и офшоринг.

Существует множество других тем, которые не рассматриваются, включая бортовую электронную аппаратуру, электронные полетные планшеты и защиту программного обеспечения. Эти темы связаны с ПО и время от времени упоминаются в тексте. Однако из-за ограничений по объему и времени они не рассматриваются. Некоторые из этих тем, впрочем, планируется осветить в новом издании *Digital Avionics Handbook* издательства CRC Press.

Литература

1. K. S. Mendis, Software safety and its relation to software quality assurance, ed. G. G. Schulmeyer, *Software Quality Assurance Handbook*, 4th edn., Chapter 9 (Norwood, MA: Artech House, 2008).
2. IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std-610-1990 (Los Alamitos, CA: IEEE Computer Society Press, 1990).
3. National Aeronautics and Space Administration, *Software Safety Standard*, NASA-STD-8719.13B (Washington, DC: NASA, July 2004).
4. B. A. O'Connell, Achieving fault tolerance via robust partitioning and N-modular redundancy, Master's thesis (Cambridge, MA: Massachusetts Institute of Technology, February 2007).
5. L. R. Wiener, *Digital Woes: Why We Should Not Depend on Software* (Reading, MA: Addison-Wesley, 1993).

*Для тех, кто не знает, воздушные тормоза находятся на крыльях.

Часть II. Контекст разработки программного обеспечения, критичного по безопасности

Часть II дает обзор того, как программное обеспечение вписывается в общий процесс разработки системы. Для того чтобы успешно реализовать *безопасное ПО*, сначала необходимо понять его роль в системе. Сосредоточиться только на ПО, не рассматривая систему и ее характеристики безопасности, это примерно то же самое, что лечить симптомы серьезной болезни, не добираясь до ее первопричины. По моему опыту, при разработке ПО, критичного по безопасности, есть пять ключевых факторов, которые должны прорабатываться глубоко и постоянно:

1. Хорошо документированная системная архитектура и определение требований.

Системная архитектура и требования должны быть ориентированы на безопасность и должны быть зафиксированы письменно. Практически невозможно успешно реализовать недокументированные требования. Чтобы убедиться, что это именно правильные требования, системные требования также должны быть валидированы на корректность и полноту.

2. Надежные практики безопасности на всех уровнях разработки.

На протяжении всей разработки потенциальные опасности и риски должны выявляться и устраняться. Безопасность должна быть неотъемлемой частью всех процессов на всех уровнях разработки, а не только обязанностью подразделения по безопасности.

3. Дисциплинированная реализация.

Документированные требования и атрибуты безопасности должны быть реализованы точно. Это включает и подтверждение того, что требования реализованы, и недопущение добавления непреднамеренной функциональности. Хорошо определенный процесс управления изменениями необходим для успешной реализации и итеративной разработки.

4. Хорошо квалифицированный персонал.

Системы, критичные по безопасности, создаются людьми. Персонал должен знать предметную область, безопасность и технологию, используемую для реализации системы. Люди, работающие в сфере критичной по безопасности разработки, должны быть лучшими из лучших, перфекционистами, которые стремятся к 100% в каждой задаче. Системы, критичные по безопасности, требуют приверженности *совершенству*, а не просто подхода *и так сойдет*.

5. Тщательное тестирование на всех уровнях.

У рассмотрений и анализов, безусловно, есть своя роль, но они не могут заменить необходимость доказывать функциональность и обоснованность утверждений о безопасности на интегрированном программном и аппаратном обеспечении.

Эта книга концентрируется на программном обеспечении, но всегда следует помнить, что ПО – лишь один из аспектов общей системы. Пять факторов, упомянутых выше, должны учитываться на протяжении всей разработки системы. Эта Часть (II), посвященная системной разработке и безопасности, задает контекст для последующих Частей (III-V), где будут подробно рассмотрены процессы разработки ПО и сопутствующие процессы, а также специальные темы, связанные с ПО.

Глава 2. Программное обеспечение в контексте системы

Сокращения

Сокращение	Расшифровка
AC	Advisory Circular (консультативный циркуляр)
ARP	Aerospace Recommended Practice (рекомендуемая аэрокосмическая практика)
DAL	Development assurance level (уровень гарантии разработки)
EASIS	Electronic Architecture and System Engineering for Integrated Safety Systems (электронная архитектура и системная инженерия для интегрированных безопасных систем)
EUROCAE	European Organization for Civil Aviation Equipment (Европейская организация по оборудованию для гражданской авиации)
FAA	Federal Aviation Administration (Федеральное управление гражданской авиации США)
PSSA	Preliminary system safety assessment (предварительная оценка безопасности системы)
RAM	Random access memory (оперативная память)
RBT	Requirements-based test (тестирование на основе требований)
ROM	Read only memory (постоянная память)
SAE	Society of Automotive Engineers (Общество автомобильных инженеров)
TCAS	Traffic collision and avoidance system (система предупреждения столкновений в воздухе)

2.1 Обзор системной разработки

Прежде чем погружаться в подробности того, как разрабатывать программное обеспечение, критичное по безопасности, важно понять контекст ПО в общей системе. ПО действует в контексте системы, а безопасность является свойством системы в целом.

Программное обеспечение само по себе не является ни безопасным, ни небезопасным. Однако ПО влияет на безопасность в самых разных типах систем, включая авиационные системы, автомобильные системы, ядерные системы и медицинские системы. Чтобы разработать ПО, которое повышает безопасность, а не ухудшает ее, сначала необходимо понять систему, в которой это ПО работает, и общий процесс обеспечения безопасности системы. В этой главе рассматривается процесс разработки систем, критичных по безопасности. В главе 3 рассматривается процесс оценки безопасности системы, который является частью общего жизненного цикла системной разработки. Обзор системной разработки и оценки безопасности дается с точки зрения авиации; однако эти концепции могут быть перенесены и в другие области, критичные по безопасности.

Воздушное судно – это крупная система, состоящая из множества систем; по сути это система систем. Системы воздушного судна могут включать навигацию, связь, шасси, управление полетом, индикацию, предотвращение столкновений, управление средой, бортовые развлекательные системы, электропитание, управление двигателем, руление на земле, реверс тяги, топливную систему, воздушные параметры и многое другое. Каждая система влияет на общую работу воздушного судна. Очевидно, что некоторые из них влияют на безопасность сильнее, чем другие. По этой причине нормы гражданской авиации и поддерживающие консультативные материалы классифицируют категории отказов в соответствии с риском. Используются пять категорий условий отказа: *catastrophic* (катастрофическое), *hazardous/severe-major* (опасное/тяжелое серьезное), *major* (серьезное), *minor* (незначительное) и *no effect* (без влияния) (эти категории рассматриваются в главе 3). Каждая функция в каждой системе, а также сочетание функциональности оцениваются с точки зрения их общего влияния на безопасную эксплуатацию воздушного судна. Эксплуатация воздушного судна включает несколько фаз, таких как руление, взлет, набор высоты, крейсерский полет, снижение и посадка. Безопасность системы оценивается для каждой фазы, поскольку последствия условий отказа могут сильно различаться в зависимости от фазы.

Разработка безопасной системы обычно требует многолетней предметной экспертизы в отношении конкретного типа системы, а также глубокого понимания того, как работают воздушное судно и его системы и как они взаимодействуют друг с другом и с людьми-операторами. В последние годы меня тревожит то, что некоторые организации, похоже, считают, будто любой человек с инженерным дипломом может разрабатывать авиационную систему. Я не против глобализации и прибыли, но формирование ключевой экспертизы и компетентности, необходимых для создания систем, критичных по безопасности, требует значительного времени, ресурсов и добросовестности.

Для гражданской авиации разработчикам рекомендуется использовать документ SAE Aerospace Recommended Practice (ARP) 4754 revision A (далее *ARP4754A*) под названием

Guidelines for Development of Civil Aircraft and Systems. ARP4754 был опубликован SAE (ранее Society of Automotive Engineers) в 1996 году и разработан командой производителей воздушных судов и авионики с участием сертифицирующих органов. EUROCAE (European Organization for Civil Aviation Equipment) также опубликовала эквивалентный документ ED-79; в российской нормативной базе этому набору системных рекомендаций соответствует Р-4754А. В декабре 2010 года SAE и EUROCAE опубликовали обновление этой рекомендуемой практики: соответственно ARP4754А и ED-79А. 30 сентября 2011 года Федеральное управление гражданской авиации США (FAA) опубликовало Advisory Circular (AC) 20-174 под названием *Development of civil aircraft systems*. AC 20-174 признает ARP4754А как “приемлемый метод установления процесса гарантии разработки”. ARP4754А делит процесс системной разработки на шесть фаз: планирование, разработка функций воздушного судна, распределение функций воздушного судна по системам, разработка системной архитектуры, распределение системных требований по элементам (включая распределение на программное и аппаратное обеспечение) и реализация системы (включая разработку программного и аппаратного обеспечения) [1]. Кроме того, в ARP4754А определены следующие интегральные процессы: оценка безопасности, назначение уровней гарантии разработки, сбор требований, валидация требований, верификация реализации, управление конфигурацией, гарантия процесса и взаимодействие с сертифицирующим органом [1]. Область действия ARP4754А включает системную разработку вплоть до завершения сертификации; однако, помимо фаз системы, описанных в ARP4754А, системная разработка включает мероприятия до запуска продукта (например, определение потребностей и проработка концепции) и после выдачи сертификата (например, производство и изготовление, поставка и ввод в эксплуатацию, эксплуатация, техническая поддержка и сопровождение, а также вывод из эксплуатации или утилизация) [2,3]. На рисунке 2.1 показана общая структура системной разработки, причем процессы ARP4754А выделены серым цветом.

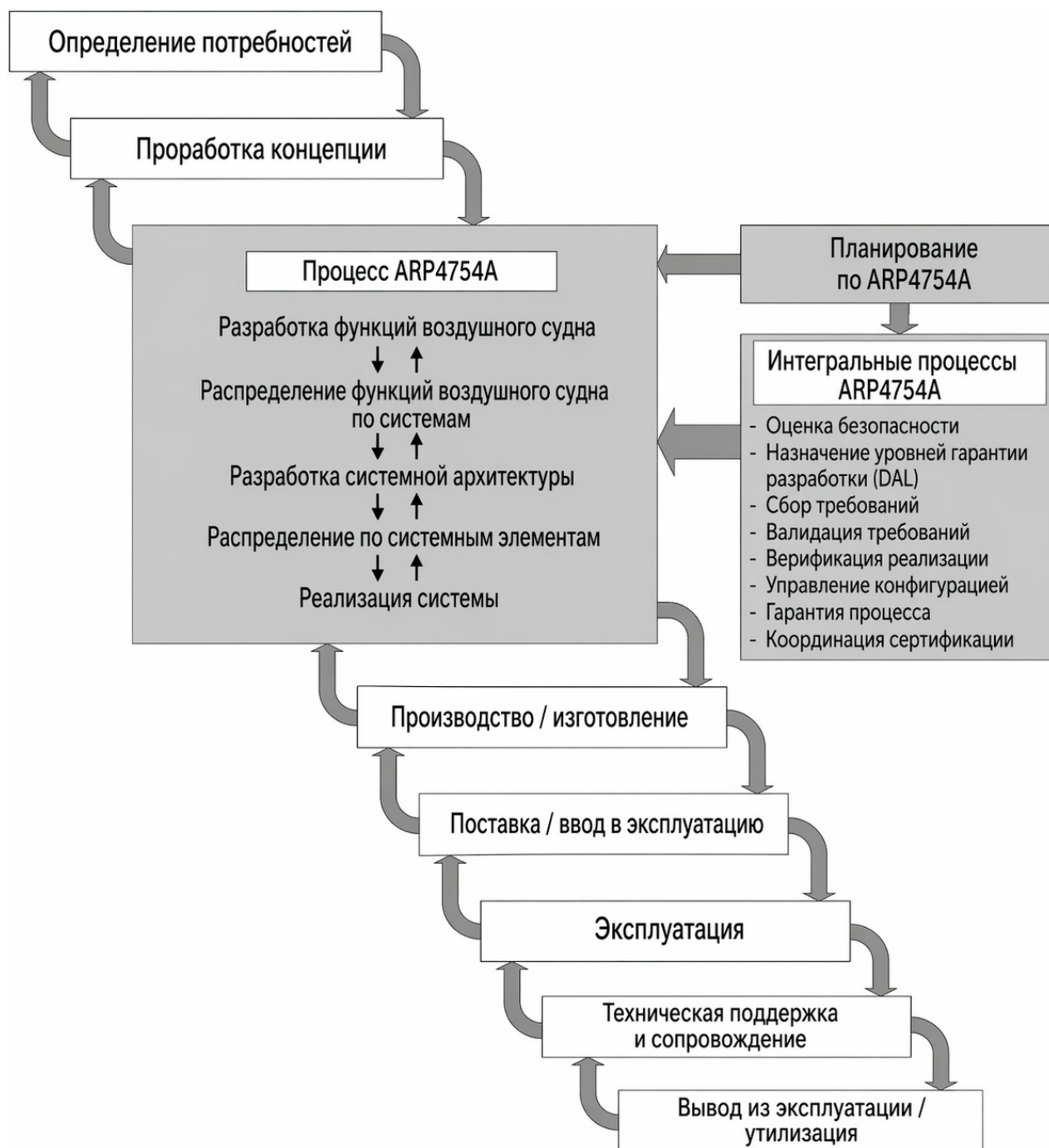


Рисунок 2.1 Общий процесс системной разработки.

Как будет обсуждаться в главе 3, процесс оценки безопасности идет параллельно процессу системной разработки. По мере того как система проходит путь от высокоуровневого проекта к реализации, аспекты безопасности оцениваются, чтобы убедиться, что система удовлетворяет требуемым уровням безопасности. Хотя ARP4754A предлагает нисходящий подход, в нем признается, что большинство систем разрабатываются итеративным и параллельным образом, с использованием как нисходящих, так и восходящих стратегий разработки. Большинство воздушных судов не состоят из систем, разработанных с чистого листа. Напротив, воздушные суда и системы, как правило, являются *производными проектами*, которые строятся на основе уже существующей функциональности и изделий. Поэтому важно, чтобы общий жизненный цикл разработки учитывал базовые системы и формировал реалистичный

жизненный цикл, удовлетворяющий как потребностям заказчика (документируемым в виде требований), так и целям безопасности.

2.2 Системные требования

2.2.1 Важность системных требований

В главе 3 мы глубже погрузимся в процесс оценки безопасности. Пока же давайте рассмотрим некоторые важные аспекты системной разработки. Пережив множество разработок продукции, я пришла к выводу, что многие проблемы, возникающие при разработке программного обеспечения, проистекают из плохо определенных системных требований и архитектуры. У разработчиков ПО, безусловно, есть свои проблемы (они будут обсуждаться в следующих главах), но многие из проблем ПО усугубляются незрелыми, неполными, некорректными, неоднозначными и/или плохо определенными системными требованиями. По моему опыту, большинству компаний следовало бы тратить больше усилий на определение, валидацию и верификацию системных требований. Это обязательно с учетом роста сложности и критичности систем. Хотя определенный прогресс и был достигнут, необходимо гораздо больше, если мы хотим и дальше разрабатывать безопасные, надежные и заслуживающие доверия системы. Органы сертификации тоже осознают эту потребность и, в связи с официальным признанием ARP4754A, усиливают внимание к процессам системной разработки, в особенности к определению требований, их валидации и верификации реализации.

2.2.2 Типы системных требований

ARP4754A рекомендует разрабатывать следующие типы требований [1]:

Требования безопасности определяют функции, которые либо способствуют безопасности воздушного судна, либо непосредственно на нее влияют. Требования безопасности выявляются процессом оценки безопасности и включают минимальные ограничения по характеристикам как для доступности (непрерывности функции), так и для целостности (корректности поведения). Они должны быть однозначно идентифицированы и быть трассируемыми на протяжении всего жизненного цикла системы.

- *Функциональные требования* задают функциональность системы для достижения требуемых характеристик. Функциональные требования обычно представляют собой сочетание требований заказчика, эксплуатационных требований, требований к характеристикам (например, по времени, скорости, диапазону, точности и разрешающей способности), физических и установочных требований, требований к сопровождаемости, интерфейсных требований и ограничений. Функциональные требования также могут быть требованиями безопасности либо иметь последствия для безопасности, которые должны оцениваться с точки зрения влияния на безопасность.

- *Прочие требования* включают нормативные требования и производные требования. Производные требования обычно являются результатом проектных решений, принимаемых по мере развития системной архитектуры. Поскольку они могут не трассироваться на требование более высокого уровня, их необходимо валидировать и оценивать в рамках процесса обеспечения безопасности системы.

2.2.3 Характеристики хороших требований

Хорошие требования не возникают сами собой. Они требуют значительных усилий, сосредоточенности и дисциплины. Чтобы писать хорошие требования, важно понимать характеристики таких требований. Авторы требований и рецензенты должны учитывать следующие характеристики хороших системных требований:

- *Atomic* (атомарность): каждое требование должно быть одним требованием.
- *Complete* (полнота): каждое требование содержит всю необходимую информацию для определения требуемой функциональности системы. Каждое требование должно быть способно существовать само по себе, без дополнительных пояснений.
- *Concise* (краткость): каждое требование должно просто и ясно указывать, что должно быть сделано, и только что должно быть сделано. Его должно быть легко читать и понимать, даже нетехническим пользователям системы. В общем случае текстовое требование не должно содержать более 30-50 слов. Требования, представленные графически, также должны быть краткими.
- *Consistent* (согласованность): требования не должны противоречить или дублировать другие требования. Согласованные требования также используют одну и ту же терминологию по всей спецификации. Требования – не место для литературного творчества.
- *Correct* (корректность): каждое требование должно быть правильным требованием для определяемой системы. Оно должно передавать точную информацию. Этот атрибут обеспечивается усилиями по валидации требования.
- *Implementation free* (свободное от реализации): каждое требование должно указывать, что требуется, не определяя, как это реализовать. В общем случае требования не должны определять проектные решения или реализацию. Однако возможны некоторые исключения, например требования к интерфейсам или производные системные требования.
- *Necessary* (необходимость): каждое требование должно задавать существенную возможность, характеристику или фактор качества. Если требование удалить, должен возникнуть дефект.
- *Traceable* (трассируемость): каждое требование должно быть однозначно идентифици-

ровано и быть легко трассируемым вплоть до требований нижнего уровня, проекта и тестирования.

- *Unambiguous* (однозначность): каждое требование должно иметь только одну интерпретацию.
- *Verifiable* (верифицируемость): должна существовать возможность подтвердить реализацию каждого требования. Поэтому требования должны быть количественно определенными и при необходимости включать допуски. Каждое требование должно быть написано так, чтобы его можно было верифицировать путем рассмотрения, [*] анализа или испытания. [†] За редкими исключениями, если поведение нельзя наблюдать в ходе верификации, требование следует переписать. Например, отрицательные требования, как правило, не поддаются верификации и требуют переработки.
- *Viable* (жизнеспособность): каждое требование должно быть реализуемым, пригодным к использованию после реализации и полезным для построения системы в целом.

Рекомендуется стандарт требований, план управления требованиями или руководство по лучшим практикам, чтобы давать авторам требований необходимые указания. Писать хорошие требования нелегко, даже если вы точно знаете, что хотите построить. Когда я преподавала курсы по DO-178B для Федерального управления гражданской авиации США (FAA), у нас было упражнение по написанию требований. Класс делился на команды, и каждой команде давалась модель LEGO[®]. Каждая команда должна была написать требования для этой модели, разобрать ее, положить детали и написанные требования в пакет и передать его другой команде, чтобы та собрала модель по требованиям. Иногда было очень забавно смотреть на конечный результат, потому что он не был похож на исходную модель. Не каждый умеет писать хорошие требования. Более того, по моему опыту, очень немногие действительно превосходно справляются с этой задачей. Хорошие руководства, примеры и обучение, основанные на конкретной предметной области, бесценны. Если для пояснения требований используются графические представления, стандарты должны объяснять правила и ограничения этих графических представлений. Например, стандарты могут ограничивать число символов на странице, ограничивать число страниц для одной графической схемы, ограничивать глубину модели и подробно определять значение каждого символа в библиотеке (поскольку различные средства моделирования по-разному трактуют символы, а уровень понимания у разработчиков различается). В главах 5, 6 и 14 соответственно подробнее обсуждаются стандарты требований, разработка программных требований и разработка на основе моделей. Эти последующие главы сосредоточены на программных аспектах требований, но большинство предложений также применимо и к определению системных требований.

2.2.4 Аспекты системных требований, которые следует учитывать

Требования к системам, критичным по безопасности, задают многочисленные функциональные возможности. В этом разделе выделены некоторые фундаментальные концепции, которые следует учитывать при разработке системных требований для программно-интенсивных и критичных по безопасности систем.[*]

2.2.4.1 Соображения, связанные с целостностью и доступностью

Системные требования должны учитывать как целостность (правильность функциональности и работы), так и доступность (непрерывность функционирования). ARP4754A определяет *integrity* и *availability* следующим образом [1]:

- *Integrity* (целостность, корректность работы): “Качественный или количественный атрибут системы или элемента, указывающий, что на него можно полагаться в том, что он будет работать корректно. Иногда выражается через вероятность несоответствия критериям корректной работы”.
- *Availability* (доступность): “Качественный или количественный атрибут, указывающий, что система или элемент находится в работоспособном состоянии в данный момент времени. Иногда выражается через вероятность того, что система (элемент) не предоставляет свой выход (то есть недоступность)”.

Потеря целостности приводит к неправильной работе (неисправному функционированию) системы. Примеры включают вводящие в заблуждение данные на основном пилотажном индикаторе, некорректную команду разрешения конфликта от системы предупреждения столкновений в воздухе (TCAS) или резкое самопроизвольное отклонение автопилота. Чтобы проектировать с учетом целостности, часто требуется архитектурное смягчение последствий, позволяющее предотвращать и/или нейтрализовать отказы. Это может включать использование избыточности, обособления, мониторинга, разнородности или других стратегий смягчения, направленных на предотвращение нарушений целостности либо защиту от них. Кроме того, для обеспечения целостности часто применяются средства обнаружения отказов и реакции на них либо локализация отказов, и это должно быть отражено в системных требованиях.

Многие системы нуждаются в защите от потери доступности. Доступность отличается от целостности тем, что функционирование должно быть непрерывным и/или восстанавливаемым при всех предсказуемых условиях эксплуатации (включая нормальные и аномальные условия). Ключевая функциональность должна быть доступна во время и после случайных аппаратных отказов, ошибок данных или сбоев программного либо аппаратного обеспечения. Системы или функции, требующие высокой доступности, как правило, требуют оборудования с очень высокой надежностью и/или используют избыточность. Когда для

обеспечения доступности используются избыточные архитектуры, они должны быть способны выдерживать отказы и/или быстро обнаруживать и устранять их последствия. Системы, включающие избыточные архитектуры, обычно имеют требования к управлению избыточностью и управлению отказами.

Следует отметить, что некоторые проектные решения могут влиять и на целостность, и на доступность. Следует учитывать и то и другое. Например, системе может потребоваться смягчать нарушения целостности, не ухудшая доступность. В первоначальной версии ARP4754 отмечалась связь между целостностью и доступностью при рассмотрении использования разнородности: “Следует отметить, что архитектурная разнородность влияет как на целостность, так и на доступность. Поскольку увеличение целостности может быть связано с уменьшением доступности, и наоборот, конкретное применение следует анализировать с обеих точек зрения, чтобы убедиться в его пригодности” [4].

2.2.4.2 Другие аспекты системных требований

Помимо вопросов целостности и доступности, в системной спецификации следует учитывать и другие аспекты, в том числе:

1. *Требования безопасности*, то есть требования, вытекающие из функционального анализа опасностей или последующих анализов безопасности.
2. *Влияние отказоустойчивости на системы*, включая определение областей отказа. Консорциум Electronic Architecture and System Engineering for Integrated Safety Systems (EASIS) определяет *область отказа* как “набор компонентов, внутренние возмущения в которых считаются равно одним отказом, независимо от того, где эти возмущения расположены, сколько их возникает и насколько далеко они распространяются внутри области отказа” [5].
3. *Плавная деградация* (например, отказобезопасный режим), включая необходимость каких-либо областей локализации. EASIS поясняет, что “область локализации – это набор компонентов, которые могут быть неблагоприятно затронуты соответствующей неисправностью. Иными словами: область локализации определяет границу, на которой распространение отказа должно остановиться” [5]. Примеры включают обнаружение отказа автопилота до того, как начнется движение рулевых поверхностей, или маркировку отказавших данных до того, как пилот на них среагирует.
4. *Обнаружение ошибок*, например [5]:
 - а. Проверки данных: исследование данных, чтобы определить, являются ли они допустимыми, находятся ли в диапазоне, актуальны ли, не замкнуты ли накоротко, не разомкнуты ли и т.д.

- b. Сравнение избыточных данных: сопоставление выходов избыточных датчиков или вычислительных устройств.
 - c. Проверка ошибок в процессоре и памяти: например, выполнение встроенных тестов, использование сторожевых таймеров, вычисление контрольных сумм для постоянной памяти (ROM) или флеш-памяти, использование встроенных исключений процессора, выполнение проверок памяти или тестирование чтения/записи оперативной памяти (RAM).
 - d. Мониторинг обмена: выполнение проверок входящих сообщений в распределенной системе, например проверка тайм-аутов сообщений, счетчиков последовательности сообщений или контрольных сумм.
5. *Обработка обнаруженных ошибок.* Некоторые примеры обработки ошибок таковы [5]:
- a. Отключить функцию.
 - b. Перейти в деградированный режим.
 - c. Выдать индикацию пользователю.
 - d. Отправить сообщение другим системам, взаимодействующим с отказавшей системой.
 - e. Сбросить систему и вернуть ее в предыдущее состояние.
 - f. Использовать другие заслуживающие доверия данные (когда обнаружена ошибка входных данных).
 - g. Перевести канал или систему в тайм-аут либо отключить их, если ошибка обнаруживается непрерывно.
6. *Обнаружение и обработка отказов*, что охватывает то, как система реагирует на аппаратные отказы (например, на постоянный или случайный отклик).
7. *Предотвращение отказов*, что может обеспечиваться выбором высококачественных компонентов (с более высоким заявленным средним временем между отказами), защитой от воздействий окружающей среды, таких как электромагнитные помехи, указанием в требованиях некоторых запрещенных отказов и/или архитектурным устранением уязвимостей.
8. *Требования к архитектуре или проекту* (например, избыточность, проверки сравнения, схемы голосования). Хотя это несколько размывает границу между требованиями и проектом, важно выделить все, что требует учета при реализации. Распространенный способ указать проектные соображения, не определяя проект слишком рано, это сопроводительный комментарий к требованию.

9. *Требования к производственному процессу.* Эти требования документируют ключевые характеристики процессов изготовления и производства аппаратуры.
10. *Функциональные ограничения.* Любые известные или желаемые ограничения системы должны быть указаны в требованиях. Например, некоторые функции могут быть разрешены только на определенных фазах полета или в определенном диапазоне применения.
11. *Критичная функциональность.* Эти требования необходимы, чтобы предотвратить потерю критичной функциональности вследствие отказа.
12. *Разделение или защита,* чтобы обеспечивать изоляцию от отказов или локализацию отказов. Разделение или защита также могут использоваться для снижения объема работ по валидации и верификации, поскольку менее критичные функции требуют меньшей строгости валидации и верификации. Разделение и защита обсуждаются в главе 21.
13. *Разнородность,* которая может использоваться для предотвращения общих отказов в двух или более элементах. Примеры разнородности включают различные проекты, алгоритмы, инструменты (например, компиляторы) или технологии [5].
14. *Избыточность,* которая может использоваться для достижения отказоустойчивости или предотвращения единственной точки отказа. Избыточные компоненты включают несколько процессоров, датчиков и т.д.
15. *Требования, связанные со временем,* чтобы учитывать такие вопросы, как время отклика в кабине экипажа, времена обнаружения отказов и реакции на них, время включения питания, время перезапуска и т.д.
16. *Последовательности,* включая требования к инициализации, холодному старту, тепловому старту и выключению питания.
17. *Переходы режимов и состояний,* такие как фазы полета, решения о включении и отключении, выборы режима и т.д.
18. *Уровни квалификации по условиям окружающей среды,* чтобы отразить требуемую стойкость к внешним воздействиям. Эти уровни зависят от критичности системы, ожидаемой среды установки и нормативных требований. RTCA/DO-160[*] описывает процесс квалификации по условиям окружающей среды для оборудования, устанавливаемого на гражданских воздушных судах.
19. *Средства для испытаний,* которые могут понадобиться для того, чтобы гарантировать корректную реализацию соображений безопасности. Примеры включают тестовые воздействия на входы, запись данных или другие интерфейсы оборудования, помогающие проведению испытаний.

20. Средства защиты от одиночных и множественных событийных сбоев, необходимые в некоторых авиационных проектах для защиты бортовых систем от радиационных эффектов, возникающих на больших высотах. Примеры распространенных решений включают непрерывные встроенные тесты, коды исправления ошибок, схемы голосования и стойкую аппаратуру.
21. Соображения надежности и сопровождаемости, особенно в поддержку допущений анализа безопасности; например, регистрация отказов и связанных с ними данных, запускаемые тесты и юстировка.
22. Требования к латентности, чтобы уменьшить вероятность того, что отказы останутся скрытыми (латентными). Примеры включают оповещения для выявления отказов и обслуживания либо интервалы встроенного тестирования.
23. Требования к пользовательскому и человеко-машинному интерфейсу, чтобы обеспечивать правильное использование системы. Существуют обширные руководящие материалы и стандарты по сертификации, посвященные потребностям человеческого фактора и минимизации эксплуатационных проблем.
24. Требования или ограничения по техническому обслуживанию, чтобы обеспечивать правильную эксплуатацию и обслуживание.
25. Требования к мониторингу безопасности, когда архитектура системы требует наличия мониторов безопасности.
26. Соображения по защищенности, включая такие меры, как шифрование, проверки конфигурации и загрузки, а также барьеры на интерфейсах.
27. Потребности будущего роста и гибкости, такие как запасы по времени и памяти, минимизация технологического устаревания и т.д.

2.2.5 Допущения в требованиях

При разработке системных требований и проекта допущения должны документироваться. Допущение – это утверждение, принцип или предпосылка, которые не являются достоверными на момент написания требований [1]. Существует множество способов документирования допущений. Некоторые часто используемые методы таковы: (1) включать их в виде комментария к требованиям или (2) создавать отдельный документ либо базу данных со ссылками на требования (например, отдельный модуль требований или отдельное поле). По мере валидации требований валидируются и допущения. Со временем допущения могут стать частью требований или проекта, а не отдельной категорией. Например, сначала можно предположить определенный интервал технического обслуживания и задокументировать это как допущение. В ходе валидации требований интервал технического обслуживания мо-

жет быть оформлен как требование (возможно, на более высоком иерархическом уровне), и тогда это уже будет не просто допущение.

2.2.6 Распределение по элементам

Обычно существует несколько иерархических уровней системных требований (например, воздушное судно, система и подсистема). Нижние уровни декомпозируются из более высоких уровней, причем на каждом более низком уровне определяется больше деталей (гранулярности). Самый нижний уровень системных требований распределяется на программное или аппаратное обеспечение для реализации. В ARP4754A программное и аппаратное обеспечение называются *items* (элементами). Затем к программным элементам применяется RTCA DO-178C (КТ-178С), а к электронным аппаратным элементам – RTCA DO-254 (КТ-254) (под названием *Design Assurance Guidance for Airborne Electronic Hardware* (Руководство по гарантии разработки бортовой электронной аппаратуры)).[*]

2.3 Валидация и верификация системных требований

Требования не только разрабатываются, но и валидируются, а их реализация верифицируется. В этом разделе рассматриваются и валидация, и верификация.

2.3.1 Валидация требований

Валидация – это процесс обеспечения того, что требования являются корректными и полными. Она может включать трассировку, анализы, испытания, рассмотрения, моделирование и/или аргументацию по аналогии [1]. Согласно ARP4754A, чем выше development assurance level (уровень гарантии разработки, DAL), тем больше методов валидации необходимо применять. В идеале системные требования, распределенные на программное обеспечение, валидируются до того, как программная команда начнет формальную реализацию. В противном случае ПО, возможно, придется перепроектировать или переписывать после того, как будут определены валидные системные требования. К сожалению, некоторые системные требования часто трудно валидировать без какого-либо работающего ПО. Прототипирование ПО – распространенный способ поддержать валидацию требований, но, как обсуждается в главе 6, с прототипированием следует обращаться осторожно. Системные требования, которые не были полностью валидированы до передачи программной команде, должны быть четко обозначены, чтобы заинтересованные стороны могли управлять риском.

2.3.2 Верификация реализации

Верификация – это процесс обеспечения того, что система реализована в соответствии со спецификацией. Она в первую очередь включает requirements-based tests (тесты, основанные на требованиях, RBT); однако также могут использоваться анализы, рассмотрения или инспекции и демонстрации. Как и в случае с валидацией требований, ARP4754A требует большего числа методов верификации для функций с более высокими DAL. Верификация

ведется непрерывно; однако формальная[*] верификация реализации выполняется после того, как аппаратное и программное обеспечение разработаны и интегрированы.

2.3.3 Рекомендации по валидации и верификации

Ниже приведены некоторые рекомендации по эффективной валидации и верификации системных требований.

Рекомендация 1: Разработайте план валидации и верификации.[†] План валидации и верификации объясняет процессы, стандарты, среду и т.д., которые будут использоваться. План должен разъяснять роли и обязанности (кто что будет делать), последовательность событий (когда это будет делаться), а также процессы и стандарты (как это будет выполняться). План также должен включать содержание, указанное в ARP4754A, и быть понятным и выполнимым для инженеров, выполняющих валидацию и верификацию.

Рекомендация 2: Обучите всех участников команды валидации и верификации. Все инженеры, отвечающие за выполнение работ по валидации и верификации, должны быть обучены планам, стандартам, процессам, методам, ожиданиям и т.д. Большинство инженеров терпят неудачу не намеренно; часто их недостатки обусловлены отсутствием понимания. Обучение помогает инженерам по валидации и верификации понять, почему определенные задачи важны и как выполнять их эффективно.

Рекомендация 3: Применяйте подходящие методы. Методы валидации сосредоточены на корректности и полноте требований и включают трассируемость, анализ, испытания, рассмотрение, моделирование и аргументацию по аналогии [1]. Методы верификации сосредоточены на корректности реализации для удовлетворения заданного набора требований; методы верификации включают рассмотрения или инспекции, анализы, испытания, демонстрации и эксплуатационный опыт [1]. Для тех областей, где испытания не выбраны в качестве метода, должно предоставляться обоснование.

Рекомендация 4: Разработайте контрольные перечни для использования командами. Контрольные перечни для валидации и верификации помогают гарантировать, что инженеры случайно ничего не забудут. Контрольные перечни не заменяют необходимость думать, но служат средствами напоминания. Заполненные контрольные перечни также дают объективные свидетельства того, что работы по валидации и верификации были выполнены.

Рекомендация 5: Проектируйте систему и требования так, чтобы их можно было испытывать. Тестопригодность (*testability*) является важным атрибутом систем, критичных по безопасности, и ее следует учитывать при сборе требований и проектировании. Примером может служить реализация средств тестирования, которые могут обнаруживать ошибки в ходе разработки и испытаний (например, bus babbler detector (средство обнаружения “болтливой” шины), partitioning violation monitor (монитор нарушения обособления), slack

checker (контроль запаса по времени), execution time monitor (монитор времени выполнения), out-of-bounds check (проверка выхода за границы), out-of-configuration check (проверка несоответствия конфигурации), built-in test (встроенный тест), журналы исключений, мониторы и т.д.).

Рекомендация 6: Разработайте и применяйте интегрированную, межфункциональную философию испытаний. Привлекайте столько дисциплин, сколько необходимо, чтобы нужные люди были вовлечены в нужное время, включая инженеров по программному обеспечению.

Рекомендация 7: Разрабатывайте и запускайте испытания рано и часто. Важны проактивная разработка и выполнение испытаний. Ранние испытания поддерживают как валидацию требований, так и верификацию реализации. При написании требований авторы должны думать о том, как будут выполняться испытания. Две практики, которые помогают: (1) привлекать тестировщиков к рассмотрению требований и принятию решений и (2) как можно раньше создать эксплуатационный системный испытательный стенд.

Рекомендация 8: Планируйте лаборатории и стенды для испытаний заранее. Испытательные средства должны быть доступны как можно раньше для использования многими дисциплинами (например, командами программного обеспечения, систем, летных испытаний и аппаратуры). На этапах планирования следует учитывать число пользователей и график испытаний, чтобы обеспечить достаточность испытательных средств (по числу и возможностям). Поскольку у заказчиков могут быть особые потребности или ожидания, рекомендуется также включать их в процесс планирования.

Рекомендация 9: Разрабатывайте эффективные функциональные или приемочные испытания и четко определяйте критерии прохождения или непрохождения. Поскольку многим инженерам на протяжении всей разработки системы потребуется использовать испытательный стенд и испытания, важно потратить время на то, чтобы сделать испытательную среду эффективной, воспроизводимой и понятной. В идеале следует автоматизировать как можно больше; однако обычно существует по меньшей мере несколько испытаний, которые невозможно автоматизировать с разумными затратами.

Рекомендация 10: Разрабатывайте интегрированную лабораторию с точными моделями для имитации входных воздействий. Чем полнее интеграционный стенд, тем меньше команда зависит от воздушного судна или средств заказчика. Чем реалистичнее имитируемые входные воздействия, тем лучше. Цель состоит в том, чтобы выявить как можно больше проблем в лаборатории и избежать необходимости искать неисправности на воздушном судне или на площадке заказчика.

Рекомендация 11: Используйте матрицы валидации и верификации. Матрицы – это простой способ убедиться, что все требования валидированы и верифицированы, а также отслежи-

вать завершение работ по валидации и верификации.

Рекомендация 12: Выполняйте испытания на робастность. Испытания на робастность обеспечивают уверенность в том, что система реагирует ожидаемым образом, когда получает неожиданные или недопустимые входные данные либо когда возникает аномальная комбинация или последовательность событий. Испытания на робастность на системном уровне часто недооцениваются и используются недостаточно. При эффективном выполнении они позволяют повысить зрелость системы и лежащего в ее основе программного обеспечения до передачи заказчику. Для систем более высокого уровня (DAL A и B) ARP4754A требует учитывать непредусмотренную функциональность в ходе верификации. Испытания на робастность дают уверенность, что система будет работать надлежащим образом даже тогда, когда события происходят не так, как ожидалось.

Рекомендация 13: Цените работу по валидации и верификации. И валидация, и верификация важны для безопасности и сертификации. Это не просто отметки или вехи в генеральном плане. Безопасность зависит от того, насколько хорошо выполнены валидация и верификация.

Рекомендация 14: Сохраняйте фокус на безопасности. Команда должна хорошо знать требования, связанные с безопасностью, и уделять им непропорционально много времени в ходе работ по валидации и верификации. Она также должна распознавать признаки преднамеренного компромисса по безопасности при испытании других функций; например, тестировщики не должны игнорировать чрезмерные значения, флаги мониторинга, исключения или сбросы.

Рекомендация 15: Помечайте проблемы по мере их возникновения. Проблемы редко исчезают сами по себе, а со временем нередко только усугубляются. Если проблема замечена, ее не следует игнорировать или скрывать. Напротив, ее следует отметить и оценить. После подтверждения, что это реальная проблема, ее следует задокументировать для дальнейшего исследования и устранения. Обычно используются журнал проблем (до введения базовой версии) и сообщения о проблемах (PR) после введения базовой версии. Проблемы из журнала, не устраненные до введения базовой версии, следует переносить в систему регистрации сообщений о проблемах, чтобы они не были упущены.

Рекомендация 16: Не жертвуйте системными испытаниями, чтобы компенсировать срывы сроков в других областях. Печальный факт жизни состоит в том, что верификация находится последним звеном цепочки и постоянно испытывает давление, вынуждающее компенсировать другие отставания по графику. Это может приводить к позднему обнаружению проблем, устранение которых обходится дорого и может серьезно испортить отношения с заказчиком.

Рекомендация 17: Координируйте работу с командами программного и аппаратного обеспечения. Координация между системными, программными и аппаратными испытаниями может приводить к более раннему обнаружению проблем (например, может быть найдена ошибка программного обеспечения, влияющая и на систему) и к более рациональному использованию ресурсов (например, некоторые системные испытания могут также верифицировать программные требования, и наоборот).

Рекомендация 18: Собирайте содержательные метрики. Метрики могут использоваться для отслеживания эффективности, сильных и слабых сторон, сроков и т.д. Сбор правильных метрик важен для эффективного управления работами по валидации и верификации. Поскольку валидация и верификация могут поглощать половину бюджета проекта[*] и критичны для сертификации, ими необходимо хорошо управлять. Метрики могут помочь, показывая, где дела идут хорошо, а где требуется дополнительное внимание.

2.4 Лучшие практики для системных инженеров

В главе 1 отмечалось, что хорошо квалифицированные инженеры необходимы для построения безопасных систем. Именно инженеры являются создателями системы. Без них система не появится. Ниже приведены рекомендуемые практики, которые команда системных инженеров должна применять, чтобы наилучшим образом использовать свои таланты и навыки. Эти практики развивают рекомендации по валидации и верификации, приведенные в разделе 2.3.3.

Лучшая практика 1: Планируйте заранее и фиксируйте это письменно. Письменные планы жизненно важны для успешной сертификации. Разрабатывайте планы по управлению проектом, безопасности, разработке, валидации и верификации, сертификации и т.д. Планы должны быть практичными, реалистичными и доступными для тех людей, которые ими пользуются.

Лучшая практика 2: Планируйте изменения. Разработайте эффективный процесс управления изменениями для всех уровней системной разработки и реализации. Процесс изменений должен быть ясно описан в планах и должен включать процесс анализа влияния изменений на системном уровне.

Лучшая практика 3: Формируйте культуру, ориентированную на безопасность. Обучайте команду вопросам безопасности и их роли в ней. Безопасность должна быть встроена в организационные процессы и входить в оценку работы команды и руководства. Кроме того, может быть полезно размещать плакаты по безопасности, приглашать специальных докладчиков для формирования сознательного отношения к безопасности и напоминать инженерам, что от их действий может зависеть безопасность их семей. Безопасность должна окрашивать все решения и действия на всех уровнях предприятия.

Лучшая практика 4: Документируйте лучшие практики. Большую ценность представляет фиксация рекомендуемых практик и дополнение их реальными примерами, основанными на многолетнем опыте. После документирования лучшие практики должны стать обязательным чтением для всех новых инженеров или участников команды. Нередко со временем лучшие практики превращаются в стандарты и процессы компании.

Лучшая практика 5: Разработайте верхнеуровневую, общеорганизационную философию проектирования. Практики на уровне организации, такие как отказобезопасные подходы, границы технического обслуживания, подходы к сообщениям (например, предупреждения, аварийные сообщения, цвет и условия появления) и т.д., должны быть задокументированы. После этого команды следует обучить этой философии, чтобы их повседневные решения соответствовали предпочтениям компании.

Лучшая практика 6: Обеспечьте, чтобы системная архитектура учитывала известные опасности из оценок опасностей воздушного судна и системы. Нередко предлагается и оценивается несколько архитектур, чтобы определить, какая из них лучше всего удовлетворяет ожиданиям по безопасности, требованиям сертификации и потребностям заказчика.

Лучшая практика 7: Разрабатывайте системную архитектуру с ранним функциональным разбиением. Как можно раньше организуйте архитектуру в логические группы и используйте ее для построения планов, оценок опасностей, требований, стандартов, отслеживания программы и т.д.

Лучшая практика 8: Выполняйте раннюю оценку безопасности. Оценка безопасности определяет жизнеспособность системной архитектуры. На ранних стадиях это не обязательно должна быть полная предварительная оценка безопасности системы (PSSA); вместо этого такая оценка используется для подтверждения концепций.

Лучшая практика 9: Фиксируйте явные требования по безопасности. Требования по безопасности должны быть задокументированы, чтобы их можно было передать командам программного и аппаратного обеспечения. Обычно требования по безопасности формулируются с помощью слова “должен” и атрибута безопасности.

Лучшая практика 10: Объясняйте обоснование требований. Полезно пояснять, откуда взялось каждое требование и как оно связано с другими требованиями, стандартами, руководствами и т.д.

Лучшая практика 11: Документируйте все допущения и обоснование производных требований. Допущения и обоснования следует документировать по мере написания требований. Не откладывайте это на потом, потому что к тому времени ход мысли может забыться.

Лучшая практика 12: Разрабатывайте proof-of-concept или prototip. Proof-of-concept

(проверка реализуемости концепции) или прототип помогает повысить зрелость требований и проекта. Однако команда должна быть готова выбросить прототип. Прототипирование – это средство достижения цели, а не сама цель. Попытка доработать прототип до уровня сертификационных стандартов обычно занимает больше времени, чем начать заново. Прототипирование подробнее рассматривается в главе 6.

Лучшая практика 13: Вовлекайте инженеров по аппаратному и программному обеспечению в процесс разработки требований. Чем раньше инженеры по аппаратному и ПО поймут систему, тем более обоснованными будут их проектные решения и тем быстрее и эффективнее они смогут реализовать систему.

Лучшая практика 14: Не считайте требования правильными, пока они не были валидированы. Лучше сохранять открытую позицию и искать способы улучшить и оптимизировать требования до завершения валидации. Несмотря на некоторые маркетинговые заявления, заказчик не всегда прав. Если что-то кажется неверным или неясным, поднимайте этот вопрос перед заказчиком как можно раньше.

Лучшая практика 15: Валидируйте требования как можно раньше. Валидация требований – это непрерывный процесс, но важно начинать его рано, привлекая правильных инженеров с правильным опытом. Раздел 2.3.3 содержит дополнительные предложения по работе по валидации.

Лучшая практика 16: Организуйте ранние и частые оценки. Ранние оценки со стороны заказчика, специалистов по человеческому фактору, пилотов, технического персонала, интеграторов систем и воздушного судна, сертифицирующих органов и т.д. помогают гарантировать, что ожидания всех заинтересованных сторон будут удовлетворены.

Лучшая практика 17: Обновляйте требования на основе входной информации. Используйте информацию от специалистов по аппаратуре, программному обеспечению, безопасности и других заинтересованных сторон для улучшения и повышения зрелости требований.

Лучшая практика 18: Проводите ранние и постоянные неформальные коллегиальные рассмотрения. Коллегиальные рассмотрения помогают рано находить проблемы и обеспечивают согласованность между участниками команды.

Лучшая практика 19: Используйте инструмент управления требованиями. Поскольку требования являются основой безопасной и успешной реализации, важно хорошо ими управлять. Инструмент управления требованиями может помочь эффективно управлять требованиями, обеспечивая более быструю и качественную реализацию системных требований. Некоторые производители используют документ Word или текстовый документ для хранения требований; однако при неаккуратном обращении такой подход может приводить к

тому, что требования не будут уникально или ясно идентифицированы, а при трассировке к требованиям или от них возникнут трудности. Это может вынудить разработчиков программного и аппаратного обеспечения интерпретировать требования и делать допущения. Иногда такие интерпретации и допущения оказываются правильными, но слишком часто это не так. При правильном использовании инструмент управления требованиями может помочь избежать части этих трудностей. Однако нужно следить за тем, чтобы при занесении требований в инструмент не потерять общий контекст и взаимосвязи между требованиями.

Лучшая практика 20: Документируйте и обеспечивайте выполнение общеорганизационных стандартов и шаблонов для системных требований и проекта. Стандарты и шаблоны требований должны поощрять качества, обсуждавшиеся ранее в этой главе. Стандарты и шаблоны проекта должны охватывать такие темы, как обоснование проекта, ограничения, распределение, пределы, пути безопасности, инициализация, режимы и органы управления, запасы, стандарты моделирования и т.д.

Лучшая практика 21: Используйте графические представления там, где это уместно. Графика (например, пользовательские интерфейсы, графика дисплеев и модели систем управления) помогает эффективно передавать концепции; однако она должна сопровождаться соответствующим текстом.

Лучшая практика 22: Проводите формальные рассмотрения. Формальные рассмотрения выполняются в ключевых точках перехода на протяжении всей разработки системы. Глава 6 обсуждает процесс коллегиального рассмотрения.

Лучшая практика 23: Обеспечивайте обучение системной команды (команд). Постоянное обучение важно для системных инженеров. Рекомендуемые темы обучения включают: написание требований, стандарты компании, предметную область или контекст (например, проектирование воздушных судов или авионики), архитектуру, атрибуты безопасности, методы обеспечения безопасности, философию безопасности и лучшие практики. Обучение должно включать конкретные примеры.

2.5 Связь программного обеспечения с системой

Программное обеспечение является частью реализации системы. Системная архитектура и требования определяют разработку ПО. Поэтому критически важно, чтобы системные инженеры, инженеры по ПО, специалисты по безопасности и инженеры по аппаратуре взаимодействовали друг с другом. Чем раньше и чем чаще такое взаимодействие происходит в ходе разработки, тем меньше будет проблем. Руководители по системам, безопасности, ПО и аппаратуре должны выстроить тесные рабочие отношения на протяжении всего проекта. Мне нравится сравнивать эту четырехдисциплинарную команду с четырьмя ножками стула. Без одной ножки стул сильно ограничен. Без двух ножек он бесполезен. Сбои в ком-

муникации между четырьмя дисциплинами ведут к ненужным трудностям, потенциально вызывая неверную интерпретацию и реализацию системных требований, пропущенные зависимости, недостаточные испытания и напрасную трату времени и денег. Во время недавней оценки испытательных данных для критичного по безопасности приложения авионики наша команда заметила, что некоторые программные требования не были испытаны. Руководитель программных испытаний сказал, что эти требования проверяет команда системных испытаний. Когда мы обратились к руководителю системных испытаний, он ответил: «Мы это не испытываем, потому что этим занимаются программисты». Неспособность общаться и совместно использовать матрицы испытаний привела к ситуации, в которой некоторые системные функции не испытывал вообще никто. Чтобы избежать этой проблемы, требовалось всего лишь простое взаимодействие.

Ниже приведены некоторые предложения, помогающие выстроить интегрированные отношения между командами системной разработки, безопасности, программного и аппаратного обеспечения.

- Размещайте команды совместно.
- Сформируйте основную группу, включающую ключевых заинтересованных лиц от каждой дисциплины. Эта группа обеспечивает руководство, принимает ключевые решения, поддерживает согласованность, оценивает и утверждает изменения и т.д.
- Работайте в этой основной группе совместно, чтобы определить и задокументировать общую философию проектирования. Затем донесите эту философию до всей команды. Координируйте передачу данных и переходы между дисциплинами (то есть обеспечивайте, чтобы данные были достаточно зрелыми для использования следующей командой).
- Поощряйте открытую и свободную от негативной реакции среду для выявления, эскалации и разрешения проблем.
- Не бойтесь вовлекать заказчика и сертифицирующий орган, когда это необходимо. Получение разъяснений по неопределенным вопросам может предотвратить неверные решения.
- Поощряйте внерабочие мероприятия для улучшения командной динамики (совместные обеды, ужины, вечеринки, выездные встречи, спортивные мероприятия и т.д.). Часто делитесь извлеченными уроками на протяжении всей программы.
- Поощряйте перекрестное обучение между командами по ключевым дисциплинарным областям (например, организуйте еженедельные или ежемесячные презентации).
- Организуйте обучение от представителей по безопасности по результатам оценки без-

опасности системы, чтобы все были знакомы с факторами, определяющими безопасность.

- Используйте инкрементное и итеративное моделирование; это помогает поэтапно наращивать систему и способствует частой координации.
- Поощряйте команды встраивать безопасность и качество в продукт. Добавить их задним числом практически невозможно.
- Отслеживайте метрики, которые стимулируют команды учитывать работу друг друга и помогать друг другу (это может включать, например, измерение количества программных функций, успешно интегрированных на аппаратуре, а не только завершение аппаратной или программной части).

Литература

1. SAE ARP4754A, *Guidelines for Development of Civil Aircraft and Systems* (Warrendale, PA: SAE Aerospace, December 2010).
2. ISO/IEC 15288, *Systems and Software Engineering-System Life Cycle Processes*, 2nd edn. (Geneva, Switzerland: International Standard, February 2008).
3. ISO/IEC 12207, *Systems and Software Engineering-Software Life Cycle Processes*, 2nd edn. (Geneva, Switzerland: International Standard, February 2008).
4. SAE ARP4754, *Certification Considerations for Highly-Integrated or Complex Aircraft Systems* (Warrendale, PA: SAE Aerospace, November 1996).
5. Electronic Architecture and System Engineering for Integrated Safety Systems (EASIS) Consortium, *Guidelines for Establishing Dependability Requirements and Performing Hazard Analysis*, Deliverable D3.2 Part 1 (Version 2.0, November 2006).

- *Reviews* в некоторых источниках также называются *inspections*.

† *Tests* в некоторых источниках также называются *demonstrations*. Для большинства сертифицирующих органов тесты, как правило, являются предпочтительным методом верификации.

*(Пожалуйста, обратите внимание, что этот раздел представляет собой лишь обзор и не охватывает тему исчерпывающим образом.)

*[] указывает на последнюю редакцию или редакцию, требуемую базисом сертификации.

*В настоящее время не до конца ясно, какое руководство применяется к аппаратным элементам, для которых DO-254 не применяется (например, к неэлектронной аппаратуре или простой аппаратуре). Ожидается, что эта область неоднозначности будет прояснена в буду-

щем при обновлении DO-254 до DO-254A. На данный момент для покрытия неоднозначных областей используются ARP4754A, DO-254 или дополнительные руководящие материалы сертифицирующих органов (например, FAA Order 8110.105, European Aviation Safety Agency Certification Memo CM-SWCEH-001 или документы по конкретному проекту Issue Papers).

*Формальная верификация используется для получения сертификационного зачета.

†Планирование валидации и верификации может оформляться в отдельных планах.

*Многие руководители проектов указывают, что более половины их бюджета и времени тратится на валидацию и верификацию.

Глава 3. Программное обеспечение в контексте оценки безопасности системы

Сокращения

Сокращение	Расшифровка
ASA	Aircraft safety assessment (оценка безопасности воздушного судна)
CCA	Common cause analysis (анализ общих причин)
CMA	Common mode analysis (анализ общих видов отказа)
CNS/ATM	Communications, navigation, surveillance, and air traffic management (связь, навигация, наблюдение и организация воздушного движения)
FDAL	Functional development assurance level (функциональный уровень гарантии разработки)
FMA	Functional hazard assessment (оценка функциональной опасности)
HIRF	High intensity radiated fields (поля высокой интенсивности излучения)
IDAL	Item development assurance level (уровень гарантии разработки элемента)
PASA	Preliminary aircraft safety assessment (предварительная оценка безопасности воздушного судна)
PRA	Particular risk analysis (анализ особых рисков)
PSSA	Preliminary system safety assessment (предварительная оценка безопасности системы)
SFHA	System functional hazard assessment (оценка функциональных опасностей системы)
SSA	System safety assessment (оценка безопасности системы)
ZSA	Zonal safety analysis (зональный анализ безопасности)

3.1 Обзор процесса оценки безопасности воздушного судна и системы

Как обсуждалось в главе 2, безопасность оценивается параллельно с разработкой воздушного судна и системы. Эта глава дает обзор процесса оценки безопасности в гражданской авиации, а также объясняет, как программное обеспечение вписывается в систему и в общую структуру работ по безопасности. На рисунке 3.1 показан итерационный характер

разработки системы и обеспечения безопасности; по мере развития системы аспекты безопасности выявляются и учитываются в проекте. Документ SAE ARP4761 под названием *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* подробно описывает подход, ожидаемый в гражданской авиации.[*]

Примечание переводчика. В российской и межгосударственной сертификационной практике в качестве аналога обычно используют Руководство Р-4761 “Руководство по методам оценки безопасности систем и бортового оборудования воздушных судов гражданской авиации”.

В других областях применяются сходные подходы (например, Military Standard 882, стандарт Министерства обороны США по системной безопасности). Далее будут описаны основные элементы процесса оценки безопасности по ARP4761.

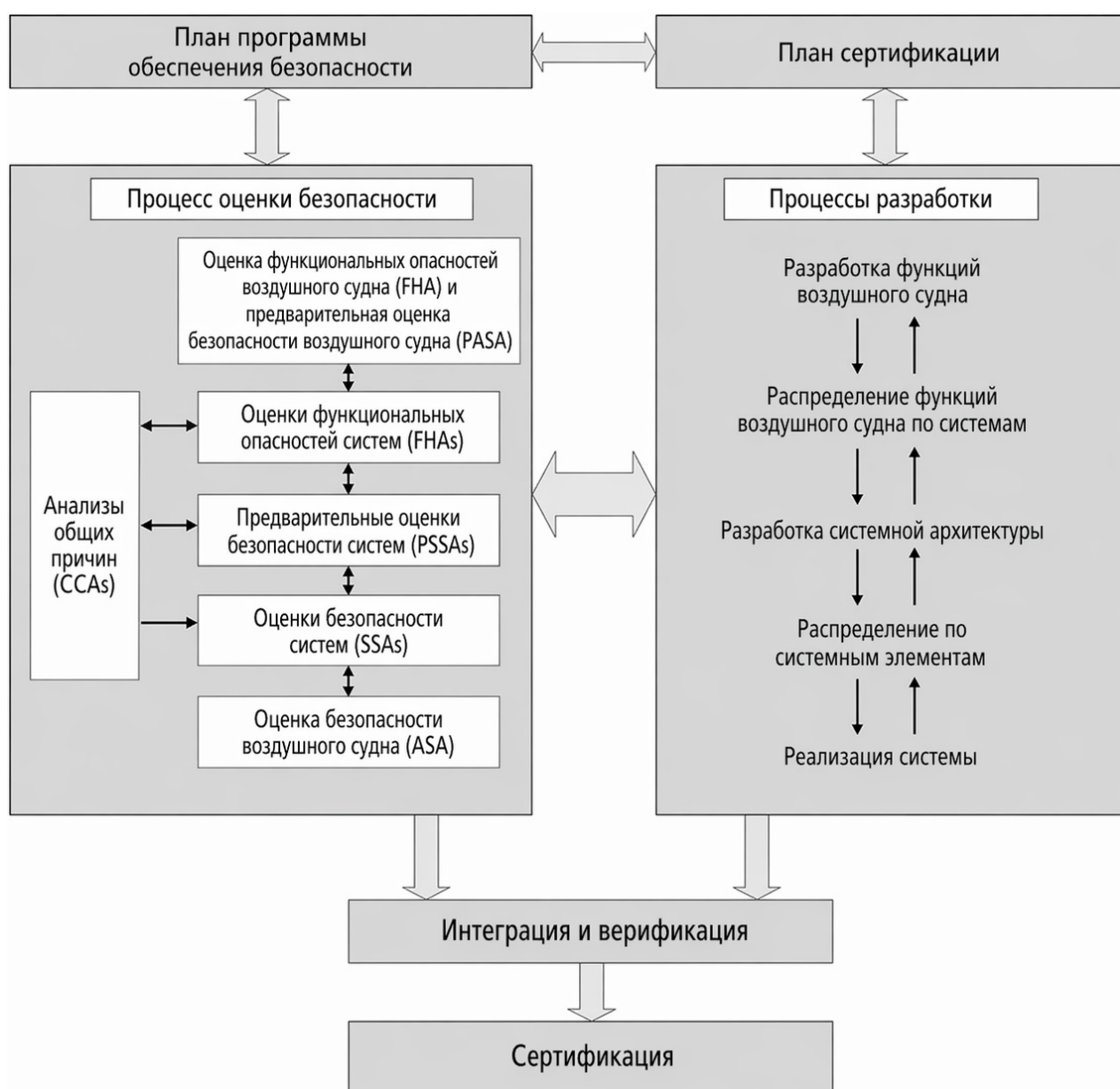


Рисунок 3.1 Обзор процессов системной разработки и оценки безопасности.

3.1.1 План программы обеспечения безопасности

Разработка плана программы обеспечения безопасности (Safety Program Plan) обычно является первой задачей, выполняемой в рамках общей оценки безопасности воздушного судна. В этом плане определяется, каким образом будет обеспечиваться безопасность воздушного судна, как будут выявляться требования, связанные с безопасностью, какие стандарты по безопасности будут применяться, какие данные будут выпускаться, какие методы будут использоваться, кто будет выполнять соответствующие задачи, каковы ключевые вехи и как различные задачи будут координироваться для обеспечения корректности, полноты и согласованности. В приложении В документа ARP4754A приведен пример плана программы обеспечения безопасности, который может использоваться как шаблон для подготовки плана безопасности конкретного воздушного судна. При его использовании пример из приложения В к ARP4754A потребуются адаптировать под конкретные потребности и организационную структуру заинтересованных сторон, участвующих в данном проекте [1]. ARP4761 содержит конкретные методики выполнения анализов, указанных в плане программы обеспечения безопасности, а также примеры и типовые контрольные перечни. Помимо плана безопасности на уровне воздушного судна, многие системы также будут иметь план безопасности на уровне отдельной системы для своей конкретной системы или изделия. На такие планы безопасности более низкого уровня могут делаться ссылки из плана программы обеспечения безопасности уровня воздушного судна. Как правило, либо план программы обеспечения безопасности, либо план сертификации (Certification Plan) уровня воздушного судна показывает взаимосвязь между различными планами и данными уровня воздушного судна и уровня отдельных систем.

3.1.2 Оценка функциональной опасности

Оценка функциональной опасности (functional hazard assessment, FMA) разрабатывается после того, как задокументированы базовая функциональность воздушного судна и концептуальный проект. FMA выявляет и классифицирует условия отказа, связанные с функциями воздушного судна и комбинациями функций. Воздействие этих условий отказа на воздушное судно оценивается относительно нормативных требований и сопутствующих руководящих материалов, которые устанавливают требования и цели по безопасности для воздушного судна. В авиации требуемые цели по безопасности различаются в зависимости от типа воздушного судна и применимых норм летной годности США, содержащихся в разделе 14 Свода федеральных нормативных актов США (Code of Federal Regulations, CFR), например: часть 25 – нормы летной годности для самолетов транспортной категории, часть 23 – нормы летной годности для самолетов нормальной категории, часть 27 – нормы летной годности для винтокрылых аппаратов нормальной категории, часть 29 – нормы летной годности для винтокрылых аппаратов транспортной категории, часть 33 – нормы летной годности для

авиационных двигателей, часть 35 – нормы летной годности для воздушных винтов.[*] Для больших самолетов транспортной категории требования, как правило, наиболее жесткие из-за общего риска для человеческой жизни. Для малых воздушных судов имеются некоторые послабления, зависящие от размера воздушного судна и области его эксплуатации. В таблице 3.1 показаны классификации тяжести вместе с последствиями условий отказа и требуемыми вероятностями для большого самолета транспортной категории, сертифицируемого по части 25 раздела 14 Свода федеральных нормативных актов США (14 CFR)[†] (там же показаны и уровни гарантии разработки, которые будут обсуждаться в разделе 3.2). Для других типов воздушных судов, двигателей и воздушных винтов используются аналогичные классификации; однако уровни вероятности для воздушных судов, сертифицируемых по части 23 раздела 14 Свода федеральных нормативных актов США (14 CFR), могут быть ниже в зависимости от класса воздушного судна. FMA является “живым” документом, который обновляется по мере выявления в ходе разработки воздушного судна дополнительных функций и условий отказа. Окончательная FMA определяет функции, условия отказа, фазы эксплуатации для каждой функции, последствия отказа функции, классификацию отказа и методы верификации.

Таблица 3.1 Тяжесть условий отказа, вероятности и уровни

Классификация тяжести	Возможный эффект условия отказа	Вероятность возникновения на один летный час (часть 25)	Уровень обеспечения
Катастрофическая	Условия отказа, которые привели бы к множественным смертельным исходам, обычно с потерей воздушного судна.	1E-9	A
Опасная/крайне серьезная	Условия отказа, которые снижали бы способность воздушного судна или способность летного экипажа справляться с неблагоприятными условиями эксплуатации до такой степени, что имелись бы значительное уменьшение запасов безопасности или функциональных возможностей; физическое истощение или чрезмерная рабочая нагрузка такого уровня, что на летный экипаж нельзя было бы рассчитывать в части точного или полного выполнения своих задач; серьезное или смертельное ранение относительно небольшого числа лиц на борту, кроме летного экипажа.	1E-7	B
Серьезная	Условия отказа, которые снижали бы способность воздушного судна или способность экипажа справляться с неблагоприятными условиями эксплуатации до такой степени, что имелись бы значительное уменьшение запасов безопасности или функциональных возможностей, значительное увеличение рабочей нагрузки экипажа или состояния, снижающие эффективность экипажа, дискомфорт для летного экипажа или физическое неблагополучие для пассажиров или бортпроводников, возможно, включая травмы.	1E-5	C
Незначительная	Условия отказа, которые не будут существенно снижать безопасность воздушного судна и при которых потребуются действия экипажа, вполне находящиеся в пределах его возможностей. Незначительные условия отказа могут включать небольшое уменьшение запасов безопасности или функциональных возможностей; небольшое увеличение рабочей нагрузки экипажа, например обычные изменения плана полета; либо некоторый физический дискомфорт для пассажиров или бортпроводников.	1E-3	D
Без влияния на безопасность	Условия отказа, которые не оказывали бы никакого влияния на безопасность; например, условия отказа, которые не влияли бы на эксплуатационные возможности воздушного судна и не увеличивали бы рабочую нагрузку экипажа.	1.0	E

Разработка FMA – это напряженный процесс. Он включает следующие мероприятия для воздушного судна в целом: (1) анализ возможных сценариев, при которых может возникнуть отказ, (2) оценку последствий, если он действительно возникнет, и (3) определение необходимости изменений. В нем используются данные предыдущих проектов, аварий и опыта летной эксплуатации. В работе участвуют опытные инженеры, пилоты, специалисты по человеческому фактору, специалисты по безопасности и представители регулирующих органов. Хотя люди обычно хорошо понимают последствия отдельной опасности, именно комбинации нескольких реалистичных отказов делают FMA особенно сложной. У каждого воздушного судна есть свои особенности, но исторические данные (например, аварии и инциденты) по сходным платформам тоже играют роль.

3.1.3 Оценка функциональных опасностей системы

Оценка функциональных опасностей системы (system functional hazard assessment, SFHA) аналогична FMA, за исключением того, что она выполняется с большей детализацией для каждой из систем воздушного судна.[*] Она анализирует архитектуру системы с целью выявления и классификации условий отказа и комбинаций условий отказа. Это может привести к обновлению FMA и общего проекта воздушного судна или системы.

3.1.4 Предварительная оценка безопасности воздушного судна

Предварительная оценка безопасности воздушного судна (PASA) представляет собой “систематическое исследование предлагаемых архитектур с целью определить, каким образом отказы могут вызывать условия отказа, выявленные в FMA” [1]. PASA является предварительным анализом безопасности самого верхнего уровня и разрабатывается на основе FMA воздушного судна. PASA, как правило, выполняется изготовителем воздушного судна и рассматривает комбинации систем и интерфейсы между ними. Входными данными для PASA являются FMA уровня воздушного судна и уровня систем, предварительные анализы общих причин (CCA), а также предлагаемые архитектуры систем и интерфейсы. PASA может выявить необходимость обновления FMA и SFHA и обычно формирует требования по безопасности более низкого уровня.

3.1.5 Предварительная оценка безопасности системы

Подобно предварительной оценке безопасности воздушного судна, предварительная оценка безопасности системы (PSSA) представляет собой исследование сверху вниз того, каким образом отказы могут приводить к функциональным опасностям, выявленным в анализе функциональной опасности и оценке функциональных опасностей системы. Если предварительная оценка безопасности воздушного судна сосредоточена на архитектуре воздушного судна и интеграции систем, то предварительная оценка безопасности системы сосредоточена на предлагаемой архитектуре конкретной системы, подсистемы или изделия. Для каждой

крупной системы воздушного судна будет существовать такая предварительная оценка безопасности системы. Для систем, состоящих из нескольких подсистем, может существовать несколько иерархических уровней такой оценки.

Как правило, такие предварительные оценки безопасности системы выполняются разработчиками систем и подробно рассматривают каждую систему воздушного судна. По мере развития систем воздушного судна они предоставляют обратную связь для предварительной оценки безопасности воздушного судна.

Процессы предварительной оценки безопасности воздушного судна и предварительной оценки безопасности системы могут приводить к необходимости дополнительной защиты или архитектурных мер смягчения последствий (например, мониторинга, обособления, резервирования или встроенного самоконтроля). Их выходные данные служат обратной связью или входными данными для анализа функциональной опасности, оценки функциональных опасностей системы, архитектуры системы и системных требований. Эти оценки также влияют на требования к программному и аппаратному обеспечению.

В целом предварительная оценка безопасности воздушного судна и предварительная оценка безопасности системы предназначены для того, чтобы показать, что предлагаемый проект будет соответствовать требованиям по безопасности, установленным нормативными требованиями, анализом функциональной опасности и оценкой функциональных опасностей системы. Эти предварительные оценки служат входными данными для итоговой, построенной по фактической реализации, оценки безопасности воздушного судна (ASA) и оценки безопасности системы (SSA). Именно они обычно задают функциональные уровни гарантии разработки для системы и уровни гарантии разработки элементов для программного обеспечения и электронной аппаратуры.

3.1.6 Анализ общих причин

Анализ общих причин (CCA) оценивает архитектуру системы и конструкцию воздушного судна с целью определить чувствительность к событиям общей причины. Это помогает обеспечить, чтобы (1) там, где это необходимо, достигалась независимость, и (2) любые зависимости были приемлемы с точки зрения требуемого уровня безопасности. Анализ общих причин подтверждает, что независимость, необходимая для безопасности и соответствия нормативным требованиям, действительно существует, либо что отсутствие такой независимости приемлемо. ARP4754A поясняет, что такой анализ “устанавливает и верифицирует требования к физическому и функциональному разделению, изоляции и независимости между системами и элементами и подтверждает, что эти требования выполнены” [1]. Его результаты используются как входные данные для предварительной оценки безопасности воздушного судна и/или предварительной оценки безопасности системы, а также

для итоговой оценки безопасности воздушного судна и/или оценки безопасности системы. Для оценки общей причины обычно выполняются три отдельных анализа; каждый из них объясняется ниже [2]:

1. *Анализ особых рисков (PRA)* оценивает “события или воздействия, находящиеся вне рассматриваемых систем и элементов, но способные нарушить заявленную независимость отказов” [2]. В ARP4761 приводится несколько примеров событий или воздействий, внешних по отношению к воздушному судну либо внешних по отношению к оцениваемой системе или элементу. Примером фактора, внешнего по отношению к воздушному судну, является столкновение с птицей. В некоторых бизнес-джетах авионика устанавливается в носовом отсеке, который на малых высотах подвержен столкновениям с птицами. Поэтому резервированные системы, такие как навигация и связь, физически размещаются по разным сторонам носового отсека, что предотвращает общую причину отказа. Другими примерами рисков, внешних по отношению к воздушному судну, являются обледенение, поля высокой интенсивности излучения (HIRF) или молния [2]. Примером фактора, внешнего по отношению к системе, является разрушение ротора двигателя, которое может пробить корпус воздушного судна, перерезав проводку или повредив резервированные блоки. Для снижения чувствительности к такому событию обычно требуется физическое разнесение электрического питания, органов управления полетом, гидравлических систем и т.д. К другим рискам, внешним по отношению к системе, относятся пожар, разрушение переборки, утечка жидкостей, злонамеренное вмешательство и отслоение протектора шины [2].
2. *Зональный анализ безопасности (ZSA)* выполняется для каждой зоны воздушного судна. Примерами таких зон являются носовой отсек, кабина экипажа, пассажирский салон, левая консоль крыла, правая консоль крыла, хвостовой отсек, топливные баки, грузовой отсек и отсек авионики. Этот анализ должен подтверждать, что установка “соответствует требованиям безопасности в части базовой установки, взаимного влияния между системами или ошибок технического обслуживания” [1]. Он проводится для каждой выявленной зоны воздушного судна и включает руководящие указания по проектированию и установке, проверку соответствия этим указаниям, а также инспекцию для выявления взаимных влияний.
3. *Анализ общих видов отказа (CMA)* рассматривает любые общие воздействия, возникающие в ходе разработки, реализации, испытаний, производства, установки, технического обслуживания или работы экипажа, которые могут привести к отказу общего вида. Такой анализ выполняется на нескольких иерархических уровнях разработки воздушного судна, начиная с верхнего уровня воздушного судна и вплоть до отдельных печатных плат. Он проверяет, что события, объединенные логическим И в дереве отка-

зов, диаграммах зависимостей и/или анализе Маркова, действительно независимы [2]. Этот анализ влияет и на разработку программного обеспечения, поскольку рассматривает такие общие виды, как ошибки разработки ПО, ошибки системных требований, функции и их мониторы, а также интерфейсы. Он помогает определить, какие уровни гарантии разработки назначаются для системы, ПО и аппаратуры. Понятие *гарантии разработки* объясняется в разделе 3.2.

3.1.7 Оценка безопасности воздушного судна и системы

В конце разработки итоговые оценки безопасности воздушного судна (ASA) и системы (SSA) подтверждают, что реализованная конструкция соответствует требованиям по безопасности, заданным анализом функциональной опасности, оценкой функциональных опасностей системы, предварительной оценкой безопасности воздушного судна и предварительной оценкой безопасности системы. В отличие от предварительных оценок, эти итоговые оценки основаны на фактически реализованной системе, а не только на предполагаемой архитектуре. Они представляют собой итоговое свидетельство того, что воздушное судно удовлетворяет нормативным требованиям по безопасности.

Оценка безопасности системы обычно разрабатывается разработчиком системы и показывает, что система удовлетворяет требованиям по безопасности, выделенным для нее в анализе функциональной опасности, оценке функциональных опасностей системы и предварительной оценке безопасности системы. Итоговая оценка безопасности воздушного судна обычно разрабатывается изготовителем воздушного судна. Она рассматривает комбинации и интеграцию систем и тесно координируется с несколькими оценками безопасности системы.

3.2 Гарантия разработки

В сложных и высокоинтегрированных электронных системах с программным обеспечением и/или программируемой аппаратурой невозможно испытать все комбинации входов и выходов, чтобы назначить вероятность отказа. Это ограничение, вместе с тем фактом, что компьютерные программы не деградируют и не отказывают со временем так, как это делают физические компоненты, приводит к необходимости концепции гарантии разработки. Гарантия разработки используется для формирования уверенности в процессе, применяемом для разработки системы и ее элементов. Гарантия разработки – это “все те запланированные и систематические действия, которые используются для подтверждения с достаточным уровнем уверенности того, что ошибки в требованиях, проекте и реализации были выявлены и исправлены таким образом, что система удовлетворяет применимой сертификационной базе” [1]. Гарантия разработки исходит из предположения, что более строгий процесс с большей вероятностью позволит выявить и устранить ошибки до поставки изделия, чем менее строгий процесс. Изначально концепция гарантии разработки была введена

применительно к ПО; однако она также применима и к другим областям, где исчерпывающее тестирование невыполнимо. В таблице 3.2 указаны области, в которых гарантия разработки применяется в гражданской авиации, а также отраслевые документы, содержащие руководящие указания для каждой из этих областей.[*]

Таблица 3.2 Документы по гарантии разработки для гражданской авиации

Обозначение	Полное название на английском	Российский эквивалентный документ	Принятое русское наименование / область применения
SAE/ARP4754A (EUROCAE/ED-79A)	<i>Guidelines for Development of Civil Aircraft and Systems</i>	P-4754A	Руководство по разработке гражданских воздушных судов и систем; системный уровень
RTCA/DO-297 (EUROCAE/ED-124)	<i>Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations</i>	P-297	Руководство по разработке и квалификации интегрированной модульной авионики; интегрированная модульная авионика
RTCA/DO-178C (EUROCAE/ED-12C)	<i>Software Considerations in Airborne Systems and Equipment Certification</i>	КТ-178С	КТ-178С, «Требования к программному обеспечению бортовой аппаратуры и систем при сертификации авиационной техники»; бортовое ПО
RTCA/DO-278A (EUROCAE/ED-109A)	<i>Guidelines for Communications, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance</i> CNS/ATM = <i>Communications, Navigation, Surveillance, and Air Traffic Management</i>	КТ-278А	КТ-278А, «Требования гарантии целостности ПО систем связи, навигации, наблюдения и организации воздушного движения (CNS/ATM)»; наземное ПО CNS/ATM
RTCA/DO-254 (EUROCAE/ED-80)	<i>Design Assurance Guidance for Airborne Electronic Hardware</i>	КТ-254	КТ-254, «Руководство по гарантии проектирования бортовой электронной аппаратуры»; бортовая электронная аппаратура
RTCA/DO-330 (EUROCAE/ED-215)	<i>Software Tool Qualification Considerations</i>	P-330	Руководство по квалификации программных инструментов; программные инструменты
RTCA/DO-331 (EUROCAE/ED-218)	<i>Model-Based Development and Verification Supplement to DO-178C and DO-278A</i>	P-331	Разработка и верификация на основе модели; дополнение к КТ-178С и КТ-278А
RTCA/DO-332 (EUROCAE/ED-217)	<i>Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A</i>	P-332	Объектно-ориентированные технологии и связанные методы; дополнение к КТ-178С и КТ-278А
RTCA/DO-333 (EUROCAE/ED-216)	<i>Formal Methods Supplement to DO-178C and DO-278A</i>	P-333	Дополнение по формальным методам к документам КТ-178С и КТ-278А
RTCA/DO-200A (EUROCAE/ED-76)	<i>Standards for Processing Aeronautical Data</i>	КТ-200А	КТ-200А, «Обработка аэронавигационных данных»; аэронавигационные данные и базы данных

Примечание переводчика. CNS/ATM расшифровывается как Communications, Navigation, Surveillance, and Air Traffic Management, то есть «связь, навигация, наблюдение и организация воздушного движения». Для ATM использовано принятое в документах ИКАО русское наименование «организация воздушного движения». Столбец «Российский эквивалентный документ» заполнен по открытым источникам; он показывает применяемые в российской практике эквивалентные документы, а не обязательно официально опубликованные переводы RTCA, SAE или EUROCAE.

Уровни гарантии разработки устанавливаются процессом оценки безопасности исходя из потенциального влияния системы на безопасность. ARP4754A определяет development

assurance level (уровень гарантии разработки) как “меру строгости, применяемую к процессу разработки с целью ограничить до приемлемого с точки зрения безопасности уровня вероятность возникновения ошибок в процессе разработки функций воздушного судна/-системы и элементов, которые имеют неблагоприятное влияние на безопасность, если эти ошибки проявятся в эксплуатации” [1]. Для большинства воздушных судов и двигателей применяются уровни гарантии разработки, показанные в таблице 3.1.[*] Вероятности для каждой категории, показанной в таблице 3.1, используются для аппаратных элементов там, где применима надежность.

3.2.1 Уровни гарантии разработки

ARP4754A выделяет две фазы системной разработки: фазу разработки функции и фазу разработки элемента. Фаза разработки функции включает разработку, валидацию, верификацию, управление конфигурацией, гарантию процесса и координацию сертификации для системы. На самом нижнем уровне системные требования распределяются на программное или аппаратное обеспечение, которые называются элементами (items). Для программных и аппаратных элементов существуют собственные фазы разработки. Руководящие указания ARP4754A применяются к фазе системной разработки; DO-178B/C применяется к фазе разработки программного обеспечения; а DO-254 (KT-254) применяется к фазе разработки электронной аппаратуры.

Традиционная V-образная модель, показанная на рисунке 3.2, демонстрирует фазы разработки функции и элемента. На уровне функции назначается функциональный уровень гарантии разработки (FDAL) на основе потенциального влияния системы на безопасность. Программному обеспечению и электронной аппаратуре назначаются уровни гарантии разработки элементов (IDAL).

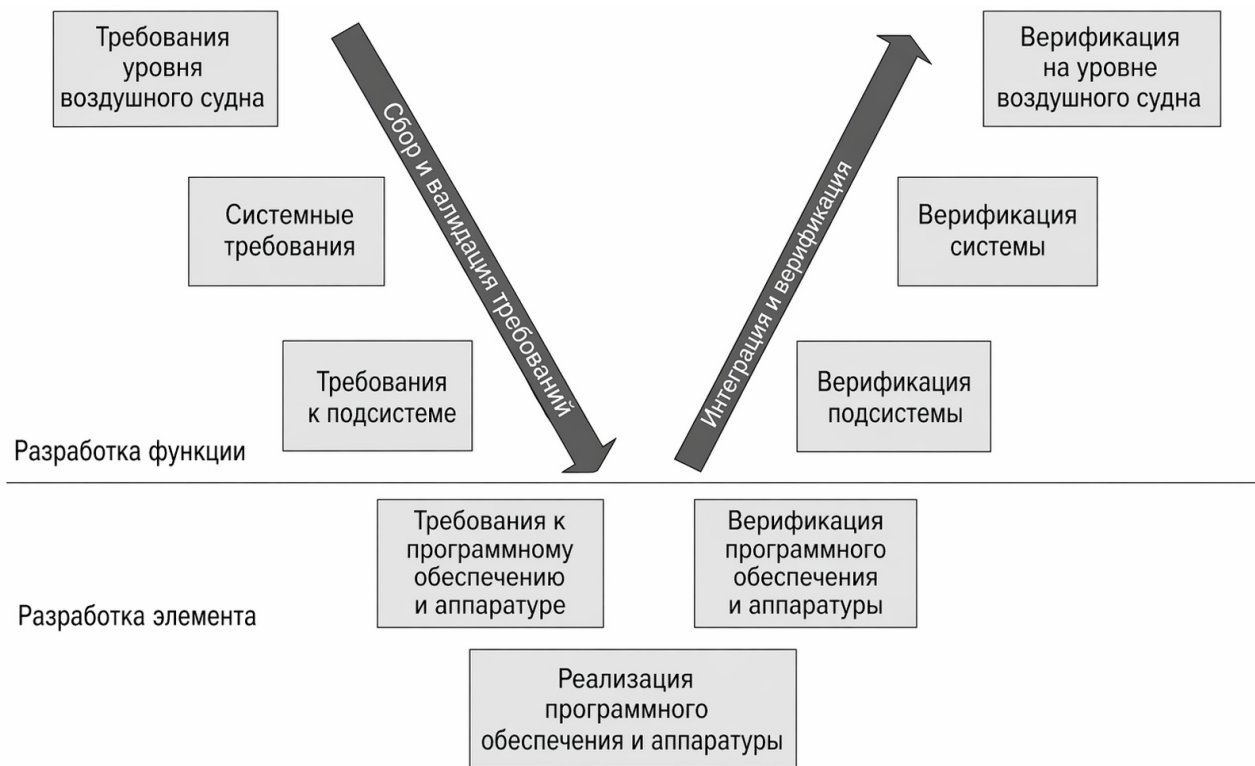


Рисунок 3.2 V-образная модель системной разработки.

Функциональный уровень гарантии разработки (FDAL) определяет объем строгости, требуемый на системном уровне (например, объем рассмотрений требований, испытаний и независимости). Уровень гарантии разработки элемента (IDAL) определяет объем строгости, требуемый при разработке конкретного элемента (программного обеспечения или электронной аппаратуры). В зависимости от архитектуры уровень гарантии разработки элемента может быть ниже, чем функциональный уровень гарантии разработки. В ARP4754A объясняется подход к их назначению. Функциональный уровень гарантии разработки определяет, какие цели ARP4754A применяются на системном уровне. Аналогично, уровень гарантии разработки элемента определяет, какие цели DO-178C (или его дополнений) применяются к ПО, а какие цели DO-254 (КТ-254) – к электронной аппаратуре.

Важно отметить, что функциональный уровень гарантии разработки и уровень гарантии разработки элемента назначаются на системном уровне и являются результатом процесса оценки безопасности. Первоначальное назначение уровня гарантии разработки элемента должно быть завершено до окончания работ по планированию в рамках DO-178C или DO-254. В DO-178C (и его дополнениях), а также далее по всей книге уровень гарантии разработки элемента для программного обеспечения называется уровнем ПО (software level). Цели DO-178C мы рассмотрим в части III книги, то есть в главах 4-12.

3.3 Как программное обеспечение вписывается в процесс обеспечения безопасности?

Роль программного обеспечения за последние 20 лет возросла. По мере того как ПО становилось все более распространенным, развивались и системные процессы, и процессы обеспечения безопасности. Традиционные методы анализа отказов и надежности для аналоговых и механических систем не работают применительно к ПО из-за его сложности и уникальности. В этом разделе рассматриваются: (1) уникальность ПО, обуславливающая необходимость процесса гарантии разработки, и (2) то, как ПО вписывается в системный процесс и процесс обеспечения безопасности.

3.3.1 Уникальность программного обеспечения

Программное обеспечение, в отличие от аппаратуры, не подчиняется законам физики: оно не изнашивается, не ломается в известных условиях и не отказывает предсказуемым образом. Некоторые другие его уникальные характеристики таковы [3]:

- Программы могут быть сложными и трудными для понимания, даже если они короткие. Программное обеспечение со временем, как правило, улучшается, поскольку скрытые ошибки обнаруживаются и исправляются.
- Исправление ошибки в одной области может внести другую ошибку, причем в области, которая на первый взгляд никак не связана с первой.
- Программное обеспечение может взаимодействовать с десятками и даже сотнями других программных модулей.
- В отличие от аппаратуры, программное обеспечение не дает предварительных признаков перед отказом. Скрытые ошибки ПО могут оставаться незаметными годами, а затем внезапно проявиться.
- Программное обеспечение можно легко изменять.
- Поиск ошибки в программном обеспечении может быть долгим и трудоемким, особенно если этим не заниматься упреждающе.

На данный момент невозможно доказать, что программное обеспечение является совершенным, или оценить, насколько близко оно к совершенству. Более того, можно пойти настолько далеко, чтобы сказать: любое ПО содержит хотя бы какие-то ошибки, потому что его создают несовершенные люди. Методы надежности, применяемые к аппаратуре, к ПО попросту неприменимы. Если бы мы полагались на традиционные аппаратные подходы к надежности, на воздушном судне было бы допустимо использовать только очень простое ПО.

3.3.2 Гарантия разработки программного обеспечения

Из-за невозможности корректно применять к программному обеспечению модели надежности концепция гарантии разработки используется в большинстве отраслей, ориентированных на безопасность, включая авиационную отрасль. Определение гарантии разработки дано в разделе 3.2. Цель гарантии разработки состоит в том, чтобы применять строгие процессы к процессу разработки для предотвращения, выявления и устранения ошибок. Чем критичнее функциональность, тем больше мероприятий по разработке и верификации применяется. Для авиационной отрасли DO-178B, а теперь и DO-178C, определяют пять уровней ПО на основе процесса обеспечения безопасности (DO-178 и DO-178A имели три уровня). Таблица 3.1 показывает связь между уровнями ПО и категориями условий отказа. ПО уровня А является наиболее строгим и относится к функциональности, которая может вызвать или внести вклад в катастрофическое событие на уровне воздушного судна. Для уровня А применяются все цели DO-178C.[*] ПО уровня Е не имеет последствий для безопасности; поэтому ни одна из целей DO-178C не требуется.

Таблица 3.3 показывает, как по мере повышения уровня программного обеспечения возрастает строгость разработки и верификации, а также число целей. С ростом уровня требуется больше мероприятий и больше независимости. Каждая из целей, суммарно представленных в таблице 3.3, подробно рассматривается в части III. Пока важно понимать следующее: чем выше уровень ПО, тем больше выполняется верификации, тем больше требуется независимости и тем больше ошибок выявляется и устраняется.

Таблица 3.3 Сводка целей DO-178C

Уровень ПО	Количество целей	Целей с независимостью	Сводка целей
Е	0	0	Никакие действия не требуются.
D	26	2	Пять планов. Разработаны требования высокого уровня. Разработана архитектура. Разработан исполняемый объектный код. Разработаны файлы компонентов параметрических данных, если они нужны, и верифицированы на корректность и полноту. Выполнены некоторые рассмотрения и анализ требований высокого уровня. Архитектура подлежит рассмотрению или анализу только при использовании обособления. Выполнены нормальные испытания и испытания на робастность требований высокого уровня. Выполнен анализ покрытия требований высокого уровня. Проведены испытания для верификации совместимости с целевым вычислителем. Выполнены мероприятия по управлению конфигурацией и гарантии качества, включая соблюдение планов и рассмотрение соответствия. Подготовлены Итоговый отчет разработки ПО и Указатель конфигурации ПО.

Уровень ПО	Количество целей	Целей с независимостью	Сводка целей
C	62	5	Все мероприятия уровня D. Три стандарта разработки. Разработаны требования низкого уровня. Разработаны данные трассируемости. Разработан исходный код. Выполнены дополнительные рассмотрения и анализ требований высокого уровня. Выполнены некоторые рассмотрения и анализ требований низкого уровня. Выполнены некоторые рассмотрения и анализ архитектуры. Выполнены рассмотрения и анализ части исходного кода. Выполнена верификация файлов компонентов параметрических данных. Выполнены нормальные испытания и испытания на робастность требований низкого уровня. Выполнен анализ покрытия требований низкого уровня. Выполнены рассмотрения процедур испытаний. Выполнены рассмотрения результатов испытаний. Выполнен анализ покрытия операторов. Выполнен анализ межкомпонентной связности по данным и по управлению. Дополнительно выполнены мероприятия по гарантии качества, включая рассмотрение планов и стандартов, проверку соблюдения стандартов и критериев перехода.
B	69	18	Все мероприятия уровней C и D. Дополнительное рассмотрение и анализ требований высокого уровня с точки зрения совместимости с целевым вычислителем. Дополнительное рассмотрение и анализ требований низкого уровня с точки зрения совместимости с целевым вычислителем и проверяемости. Дополнительное рассмотрение и анализ архитектуры с точки зрения совместимости с целевым вычислителем и проверяемости. Дополнительное рассмотрение и анализ исходного кода с точки зрения проверяемости. Выполнен анализ покрытия решений.
A	71	30	Все мероприятия уровней B, C и D. Выполнен анализ покрытия по модифицированному условию/решению (MC/DC). Выполнена верификация трассируемости от исходного кода к объектному коду.

3.3.3 Другие точки зрения

Научно строго доказать утверждения, лежащие в основе подхода гарантии разработки, практически невозможно. Кроме того, хороший процесс сам по себе не обязательно означает устойчивый и безопасный продукт. Как я часто в шутку говорю: “Не все программное обеспечение уровня A создано одинаковым”. Можно иметь превосходный процесс и все равно выпускать ПО с ошибками, часто из-за использования неопытного персонала или плохих системных требований. И наоборот, команда может иметь процесс ниже стандарта и при этом создать устойчивое и надежное ПО – главным образом благодаря использованию высококвалифицированных и опытных инженеров. DO-178C предполагает использование квалифицированных и хорошо обученных людей; однако адекватность людей трудно измерить. Некоторые сертифицирующие органы проверяют резюме и историю обучения, чтобы убедиться, что используются квалифицированные и надлежащим образом подготовленные специалисты.

Для компаний, у которых еще нет зрелой культуры работ по безопасности, может возникать соблазн просто назначить программному обеспечению тот или иной уровень, не понимая,

чем именно этот уровень обоснован. Если разработку ПО оторвать от системных требований и факторов, определяющих безопасность, это может привести к серьезным проблемам. Именно поэтому в DO-178C, ARP4754A и ARP4761 постарались точнее описать, как учитывать программную реализацию в составе системы. Но соблазн назначать уровень ПО формально, без системного обоснования, сохраняется и за этим нужно специально следить.

Поскольку гарантия разработки не может быть научно строго обоснована, есть и те, кто ставит под сомнение и даже отвергает его использование. Большинство таких критиков продвигают использование моделей безопасности и надежности на уровне программного обеспечения. Они расширяют процесс оценки безопасности системы, включая в него ПО, и минимизируют зависимость от гарантии разработки. Такие усилия используют сценарии применения, связанные с безопасностью, и модели, чтобы продемонстрировать, что система надежна при заданном наборе условий, которые, как ожидается, будут встречаться в эксплуатации. Для разработки сценариев и прогонки системы по этим сценариям требуются значительные усилия и высокая квалификация. Фокус на безопасности полезен, поскольку помогает убедиться, что проект ПО поддерживает безопасность системы. Однако на сегодняшний день методы моделирования надежности ПО все еще ограничены и остаются предметом споров.

3.3.4 Некоторые предложения по учету программного обеспечения в системном процессе обеспечения безопасности

Как уже отмечалось ранее, безопасность – это атрибут системы, а не атрибут программного обеспечения. Для безопасной реализации системы необходимы качественные системная инженерия, инженерия безопасности, аппаратная инженерия и программная инженерия. Ниже приведены некоторые предложения по улучшению связи между этими дисциплинами:

- Присваивайте программным требованиям, которые непосредственно реализуют защитные функции, специальные признаки, связанные с безопасностью, чтобы было видно, какие программные требования наиболее критичны для безопасности. Это также помогает полноценно учитывать последствия изменений для безопасности.
- Привлекайте системных инженеров и инженеров по безопасности к рассмотрению программных требований и архитектур.
- Координируйте производные программные требования с командой по безопасности сразу после их выявления, чтобы убедиться в отсутствии последствий для безопасности. Включайте в производные требования обоснование, чтобы их можно было понять, оценить с точки зрения безопасности и сопровождать.
- Размещайте команды системной инженерии, инженерии безопасности, аппаратной и

программной инженерии совместно.

- Выполняйте предварительную оценку безопасности системы на раннем этапе, чтобы определить уровни программного обеспечения и четко обозначить, какие системные функции влияют на безопасность; то есть сделать понятным, чем именно определяется уровень ПО и какие меры снижения последствий (защитные механизмы) требуются в ПО.
- Предоставляйте всей программной команде сводку по предварительной оценке безопасности системы, чтобы она понимала обоснование уровня программного обеспечения и влияние своей работы на безопасность.
- Рассмотрите возможность проведения оценки безопасности вплоть до уровня программного обеспечения в поддержку общего процесса оценки безопасности. Это особенно полезно, когда защита (например, обособление или мониторинг) реализуется в ПО. Обучайте программную команду основам системной разработки, процессу оценки безопасности системы, назначению уровней гарантии разработки, распространенным методам разработки систем, критичных по безопасности, а также тому, на что им следует обращать внимание в своей системе.
- Используйте опытные команды. Если привлекаются младшие инженеры (всем нам когда-то приходится начинать), объединяйте их в пары с опытными инженерами.
- Формируйте культуру, ориентированную на безопасность. Если высшее руководство поддерживает безопасность и подтверждает это делом, это невероятно сильно влияет на то, как безопасность воспринимается и реализуется на практике.

Литература

1. SAE ARP4754A, *Guidelines for Development of Civil Aircraft and Systems* (Warrendale, PA: SAE Aerospace, December 2010).
2. SAE ARP4761, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* (Warrendale, PA: SAE Aerospace, December 1996).
3. J. P. Bowen and M. G. Hinchey, Ten commandments of formal methods... Ten years later, *IEEE Computer*, January 2006, 40-48.

*В настоящее время ARP4761 обновляется. Когда он станет доступен, вместо ARP4761 следует использовать ARP4761A.

*Здесь имеются в виду части раздела 14 Свода федеральных нормативных актов США (Code of Federal Regulations, CFR), обычно сокращаемого как 14 CFR: часть 25 – Airworthiness Standards: Transport Category Airplanes, часть 23 – Airworthiness Standards: Normal Category

Airplanes, часть 27 – Airworthiness Standards: Normal Category Rotorcraft, часть 29 – Airworthiness Standards: Transport Category Rotorcraft, часть 33 – Airworthiness Standards: Aircraft Engines, часть 35 – Airworthiness Standards: Propellers.

†Речь идет о части 25 раздела 14 Свода федеральных нормативных актов США (Code of Federal Regulations, CFR), обычно обозначаемого как 14 CFR: Airworthiness Standards: Transport Category Airplanes.

*Как правило, для каждой системы воздушного судна существует отдельный SFHA.

*Также включены эквивалентные номера документов EUROCAE.

*Для некоторых малых воздушных судов, сертифицируемых по части 23 раздела 14 Свода федеральных нормативных актов США (14 CFR), уровни ниже. См. консультативный циркуляр FAA AC 23.1309-1[], где [] обозначает последнюю редакцию.

*Цели DO-178С объясняются в части III книги.

Часть III. Разработка программного обеспечения, критичного по безопасности, с использованием DO-178C (КТ-178С)

Документ RTCA/DO-178C (русский аналог: КТ-178С) носит название *Software Considerations in Airborne Systems and Equipment Certification* (по русскому изданию КТ-178С: Требования к программному обеспечению бортовой аппаратуры и систем при сертификации авиационной техники). Он был разработан как набор рекомендаций на основе консенсуса международного сообщества[*] при поддержке RTCA[†] и EUROCAE[‡] и опубликован 13 декабря 2011 года. Ожидается, что Федеральное управление гражданской авиации США (FAA), Европейское агентство по авиационной безопасности (EASA) и другие органы гражданской авиации вскоре признают этот документ приемлемым средством подтверждения соответствия требованиям нормативных актов.[§] DO-178C и его эквивалент EUROCAE (ED-12C) были предшествованы документами DO-178/ED-12, DO-178A/ED-12A и DO-178B/ED-12B. DO-178C дает руководство по разработке бортового ПО; однако большая часть этого руководства применима и к другим областям, критичным по безопасности. Часть III дает обзор DO-178C и рекомендации по разработке ПО в соответствии с DO-178C. Этот раздел не будет повторять то, что уже содержится в DO-178C, а даст практическую информацию о том, как применять DO-178C в реальных проектах.

Примечание переводчика. В исходном тексте книги встречались сопоставления DO-178B и DO-178C, отражавшие переходный период после выхода DO-178C. В данном переводе такие сопоставления и обещания их дальнейшего рассмотрения удалены как утратившие актуальность.

Часть III можно кратко охарактеризовать следующим образом:

- В главе 4 приводятся история DO-178, обзор DO-178C и краткое описание шести документов, опубликованных вместе с DO-178C. В главе 5 рассматривается процесс планирования по DO-178C и рекомендуются лучшие практики эффективного планирования.
- В главах 6, 7 и 8 обсуждаются соответственно разработка программных требований, проектирование и реализация. В каждой главе рассматривается, что требуется целями DO-178C, и обсуждаются рекомендации по выполнению этих целей.
- В главах 9, 10, 11 и 12 рассматриваются интегральные процессы соответственно верификации, управления конфигурацией программного обеспечения, обеспечения качества ПО и взаимодействия с сертифицирующим органом. И здесь эти главы кратко рассматривают цели DO-178C, но сосредотачиваются на практических способах их выполнения.

Главы 6 (требования) и 9 (верификация) длиннее остальных глав из-за исключительно важ-

ного значения этих двух предметных областей для безопасности и соответствия DO-178C.

*В состав совместного комитета входили представители промышленности, регулирующих органов и академической среды из США, Канады, Европы, Южной Америки и Азии.

†RTCA (ранее Radio Technical Commission for Aeronautics) является объединением правительственных и отраслевых авиационных организаций США. RTCA получает часть средств от федерального правительства, но также поддерживается ежегодными взносами членов и продажей документов.

‡EUROCAE (European Organization for Civil Aviation Equipment) является международной организацией, членство в которой открыто для европейских изготовителей и пользователей авиационного оборудования, национальных органов гражданской авиации, отраслевых ассоциаций, а при определенных условиях и для неевропейских участников.

§Такое признание обычно оформляется в виде Advisory Circular (консультативного циркуляра, AC) для FAA и эквивалентных консультативных материалов со стороны других сертифицирующих органов. Ожидается, что для признания DO-178C будет опубликован FAA AC 20-115C.

Глава 4. Обзор DO-178C и сопутствующих документов

Сокращения

Сокращение	Расшифровка
AL	Assurance level (уровень гарантии разработки)
CAST	Certification Authorities Software Team (группа органов сертификации по программному обеспечению)
KK1	Категория контроля 1 (control category #1)
KK2	Категория контроля 2 (control category #2)
CNS/ATM	Communications, navigation, surveillance, and air traffic management (связь, навигация, наблюдение / организация воздушного движения)
COTS	Готовое коммерческое ПО (commercial off-the-shelf)
DP	Discussion paper (дискуссионный документ)
EUROCAE	European Organization for Civil Aviation Equipment (Европейская организация по оборудованию для гражданской авиации)
FAQ	Frequently asked question (часто задаваемый вопрос)
OOT	Объектно-ориентированная технология (object-oriented technology)

Сокращение	Расшифровка
OOT&RT	Объектно-ориентированная технология и связанные техники (object-oriented technology and related techniques)
PDS	Ранее разработанное ПО (previously developed software)
PR	Сообщение о проблеме (problem report)
PSAA	План аспектов ПО для одобрения (Plan for Software Aspects of Approval)
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
SAS	Итоговый отчет разработки ПО (Software Accomplishment Summary)
SDD	Описание проекта ПО (Software Design Description)
SCI	Указатель конфигурации ПО (Software Configuration Index)
SCM	Управление конфигурацией ПО (software configuration management)
SCMP	План управления конфигурацией ПО (Software Configuration Management Plan)
SDP	План разработки ПО (Software Development Plan)
SECI	Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index)
SQA	Гарантия качества ПО (software quality assurance)
SWRD	Документ требований высокого уровня к ПО (Software Requirements Document)
SQAP	План гарантии качества ПО (Software Quality Assurance Plan)
SVCP	Примеры и Процедуры верификации (Software Verification Cases and Procedures)
SVR	Результаты верификации ПО (Software Verification Results)
SVP	План верификации ПО (Software Verification Plan)
RT	Related techniques (связанные техники)
SC-205	Special Committee #205 (специальный комитет N 205)
SW	Software (ПО)
TQL	Tool qualification level (уровень квалификации инструмента)
WG-71	Working Group #71 (рабочая группа N 71)

4.1 История DO-178

DO-178 и его эквивалент EUROCAE[*] (ED-12) имеют 30-летнюю историю. В таблице 4.1 кратко показана эволюция DO-178 и связанных с ним документов. В 1982 году была опубликована первая версия DO-178. Документ был очень кратким и предоставлял лишь высокоуровневую основу для разработки программного обеспечения.

Таблица 4.1 Эволюция DO-178 и связанных документов

Документ	Год публикации	Содержание	Российский эквивалентный документ
DO-178	1982	Базовые сведения по разработке бортового программного обеспечения.	Прямой эквивалент не выявлен
DO-178A	1985	Усиливает принципы программной инженерии по сравнению с DO-178; охватывает и верификацию, и валидацию требований.	Прямой эквивалент не выявлен
DO-178B	1992	Значительно подробнее DO-178A; вводит подход через цели, а не через предписанные действия; делает заметнее процессы и данные жизненного цикла; не включает валидацию требований.	КТ-178В
DO-248В	2001	Исправления и разъяснения к DO-178В, включая FAQ и discussion papers; не считается самостоятельным руководством.	Прямой эквивалент не выявлен
DO-278	2002	Применяет подход DO-178В к программному обеспечению CNS/АТМ (связь, навигация, наблюдение / организация воздушного движения) и добавляет терминологию и рекомендации для этой области.	КТ-278
DO-178С	2011	Уточняет ряд областей, добавляет рекомендации по параметрическим данным и ссылки на DO-330 для квалификации инструментов.	КТ-178С

Документ	Год публикации	Содержание	Российский эквивалентный документ
DO-278A	2011	Самостоятельный документ для CNS/ATM (связь, навигация, наблюдение / организация воздушного движения); по структуре близок к DO-178C, но содержит дополнительные изменения терминологии и рекомендации для наземного ПО CNS/ATM.	КТ-278А
DO-248C	2011	Обновляет DO-248B под редакцию DO-178C, расширяет материал по DO-278A и добавляет пояснения к целям DO-178C и его дополнениям.	Р-248С
DO-330	2011	Руководство по квалификации инструментов; самостоятельный документ, на который ссылаются DO-178C и DO-278A.	Р-330
DO-331	2011	Дополнение по разработке и верификации на основе моделей к DO-178C и DO-278A.	Р-331
DO-332	2011	Дополнение по объектно-ориентированным технологиям и связанным техникам к DO-178C и DO-278A.	Р-332
DO-333	2011	Дополнение по формальным методам к DO-178C и DO-278A.	Р-333

Примечание переводчика. Колонка с российскими эквивалентными документами добавлена для практической навигации по российской нормативной базе. Для ранних редакций семейства DO-178 и для DO-278 прямые российские эквиваленты в открытых источниках не выявлены.

В период между 1982 и 1985 годами дисциплина программной инженерии значительно повзрослела, как и сам DO-178. В 1985 году был опубликован DO-178A. Многие системы, которые существуют и сегодня, использовали DO-178A как средство подтверждения соответствия. Однако у DO-178A было несколько сложных особенностей. Во-первых, DO-178A охватывал и валидацию требований, и верификацию. DO-178 и DO-178A появились раньше

ARP4754 и ARP4761, которые сформировали основу системного процесса и процесса оценки безопасности. С появлением ARP4754 было определено, что валидация требований (подтверждение того, что выбраны правильные требования) является системным мероприятием, тогда как верификация требований (подтверждение того, что требования, распределенные на программное обеспечение, реализованы корректно) является мероприятием в области ПО. Во-вторых, DO-178A не был основан на целях, что затрудняло объективную оценку соответствия ему. В-третьих, DO-178A допускал structural testing (структурное тестирование), а не structural coverage analysis (анализ структурного покрытия). Как будет объяснено в главе 9, структурное тестирование недостаточно для выявления unintended functionality (непредусмотренной функциональности) и для измерения полноты тестирования, основанного на требованиях.

DO-178B был опубликован в конце 1992 года. Он разрабатывался параллельно с ARP4754 и ARP4761, которые были опубликованы немного позже. DO-178B содержит больше деталей, чем DO-178A, и стал *де-факто* стандартом разработки программного обеспечения в гражданской авиации. DO-178B основан на целях; он стремится определить, что разработчик должен сделать, но оставляет гибкость в том, как это будет выполнено. Документ также требует большей прозрачности того, что делается, за счет требования документированных доказательств (данных жизненного цикла) для всех мероприятий, по которым заявляется зачет для сертификации; это делает возможной объективную оценку соответствия. Кроме того, как отмечалось выше, DO-178B не охватывает валидацию требований; это ответственность системной команды. И, наконец, DO-178B значительно прояснил ожидания в отношении анализа структурного покрытия.

В 1999-2001 годах RTCA опубликовала DO-248, DO-248A и DO-248B. DO-248 начинался как ежегодный отчет, предназначенный для разъяснения некоторых наиболее сложных аспектов DO-178B. Финальная версия отчета была опубликована в 2001 году как DO-248B (ее эквивалент EUROCAE – ED-94B); она включала содержимое предыдущих выпусков. DO-248B содержал 12 errata (опечаток и мелких исправлений) к DO-178B, 76 frequently asked questions (часто задаваемых вопросов, FAQ) и 15 discussion papers (дискуссионных записок, DP). FAQ – это короткое разъяснение объемом менее одной страницы, написанное в форме вопроса и ответа. DP – это тоже разъяснение, но более длинное, чем одна страница, и изложенное в форме тематического объяснения. DO-248B никогда не считался руководящим материалом; он лишь разъяснял руководство, уже содержащееся в DO-178B. Это информационный и учебный документ, который помогает пользователям лучше понимать и применять DO-178B. Как будет объяснено далее в этой главе, DO-248B был заменен на DO-248C.

В 2005 году RTCA и EUROCAE создали совместный комитет для обновления DO-178B (и

ED-12B), чтобы определить дальнейший путь учета изменений в программных технологиях.[*] Цели комитета состояли в продолжении следующего [1,2]:

- Содействие безопасной реализации авиационного программного обеспечения
- Обеспечение четкой и последовательной связи с системными процессами и процессами оценки безопасности
- Учет новых тенденций и технологий в области программного обеспечения
- Реализация гибкого подхода к изменениям по мере развития технологий

Входными материалами для работы комитета были DO-178B, DO-278, DO-248B, публикации сертифицирующих органов (включая политику, руководства, issue papers, документы Certification Authorities Software Team [CAST][*], исследовательские отчеты и т.д.), техническое задание комитета, постоянно пополняемый перечень нерешенных вопросов, а также опыт и экспертиза сотен специалистов. Результатом стали семь документов, которые часто называют *green covers* (зелеными обложками), поскольку печатная версия каждого документа RTCA издается в зеленой обложке. В конце 2011 года RTCA опубликовала следующие документы (неплохой рождественский подарок для тех из нас, кто в составе комитета посвятил этому 6,5 лет жизни):

- DO-178C/ED-12C, *Software Considerations in Airborne Systems and Equipment Certification*
- DO-278A/ED-109A, *Guidelines for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance*
- DO-248C/ED-94C, *Supporting Information for DO-178C and DO-278A*
- DO-330/ED-215, *Software Tool Qualification Considerations*
- DO-331/ED-218, *Model-Based Development and Verification Supplement to DO-178C and DO-278A*
- DO-332/ED-217, *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*
- DO-333/ED-216, *Formal Methods Supplement to DO-178C and DO-278A*

На рисунке 4.1 показана взаимосвязь семи документов. Далее приводится краткое описание каждого документа; более подробно они будут объяснены в последующих главах. Документы RTCA серии DO и документы EUROCAE серии ED идентичны, за исключением макета страниц (RTCA использует формат letter, тогда как EUROCAE использует формат A4) и добавленного французского перевода в документах EUROCAE. В остальной части этой книги будут использоваться только номера RTCA.

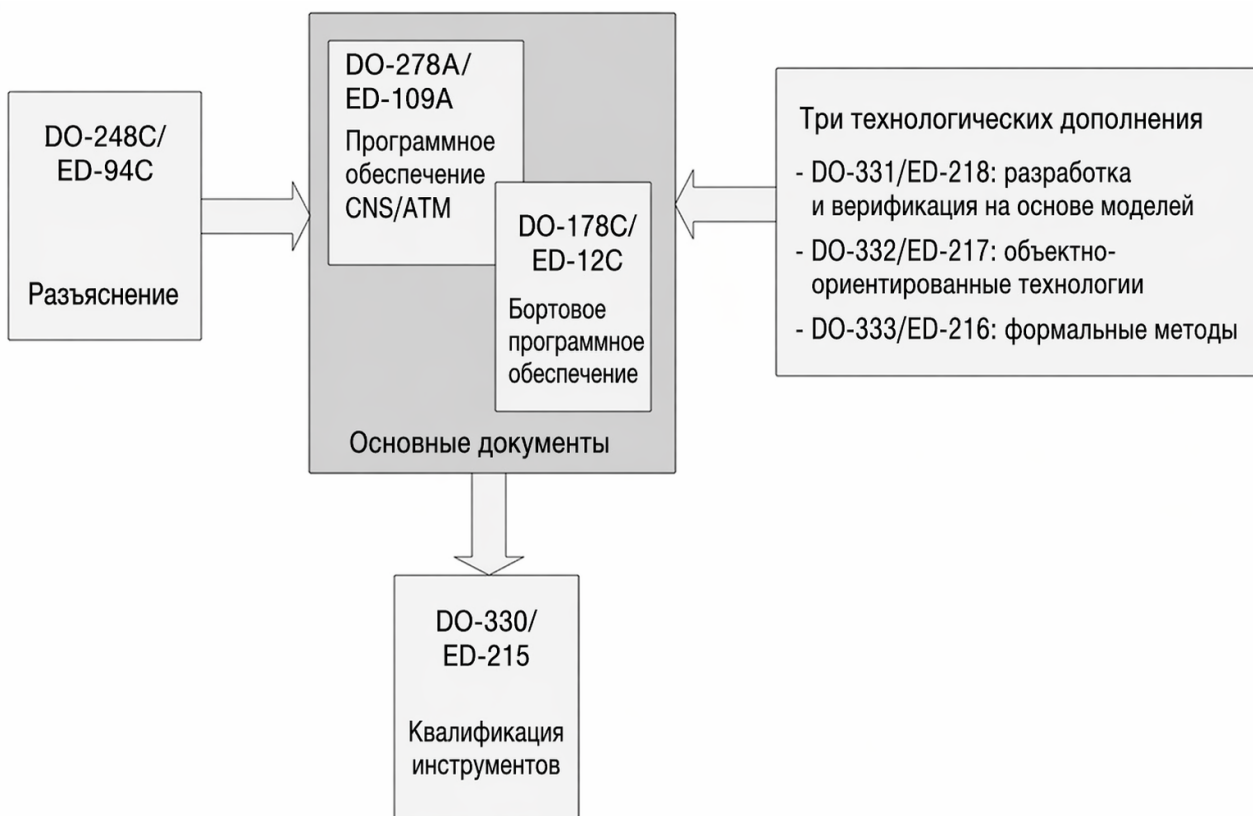


Рисунок 4.1 Взаимосвязь DO-178C и связанных документов.

4.2 Основные документы DO-178C и DO-278A

DO-178C и DO-278A практически одинаковы, за исключением того, что DO-178C ориентирован на бортовое программное обеспечение, а DO-278A – на наземное ПО CNS/ATM (связь, навигация, наблюдение / организация воздушного движения). В разделе 4.2.1 кратко суммированы основные различия между этими двумя документами. Поскольку DO-178C и DO-278A почти одинаковы, а сообщество пользователей у DO-178C больше, чем у DO-278A, после раздела 4.2.1 далее будет упоминаться только DO-178C, если нет заметного отличия.

DO-178C и DO-278A часто называют *основными документами*, поскольку именно они задают базовые концепции и организацию, на которых построены остальные документы. Эти основные документы задуманы максимально независимыми от конкретных технологий. DO-248C и дополнения ссылаются на эти два основных документа и дают детали, специфичные для конкретных технологий.

DO-178C был обновлен, чтобы устранить неясные аспекты прежнего руководства, согласоваться с системными документами и документами по безопасности (ARP4754A и ARP4761) и получить полный набор таблиц целей, согласованный с вводной частью документа. DO-178C состоит из 12 разделов, приложения Annex A, которое суммирует цели основной части, приложения Annex B, содержащего глоссарий, и нескольких вспомогательных приложений. Взаимосвязь разделов DO-178C показана на рисунке 4.2 и кратко описана ниже:

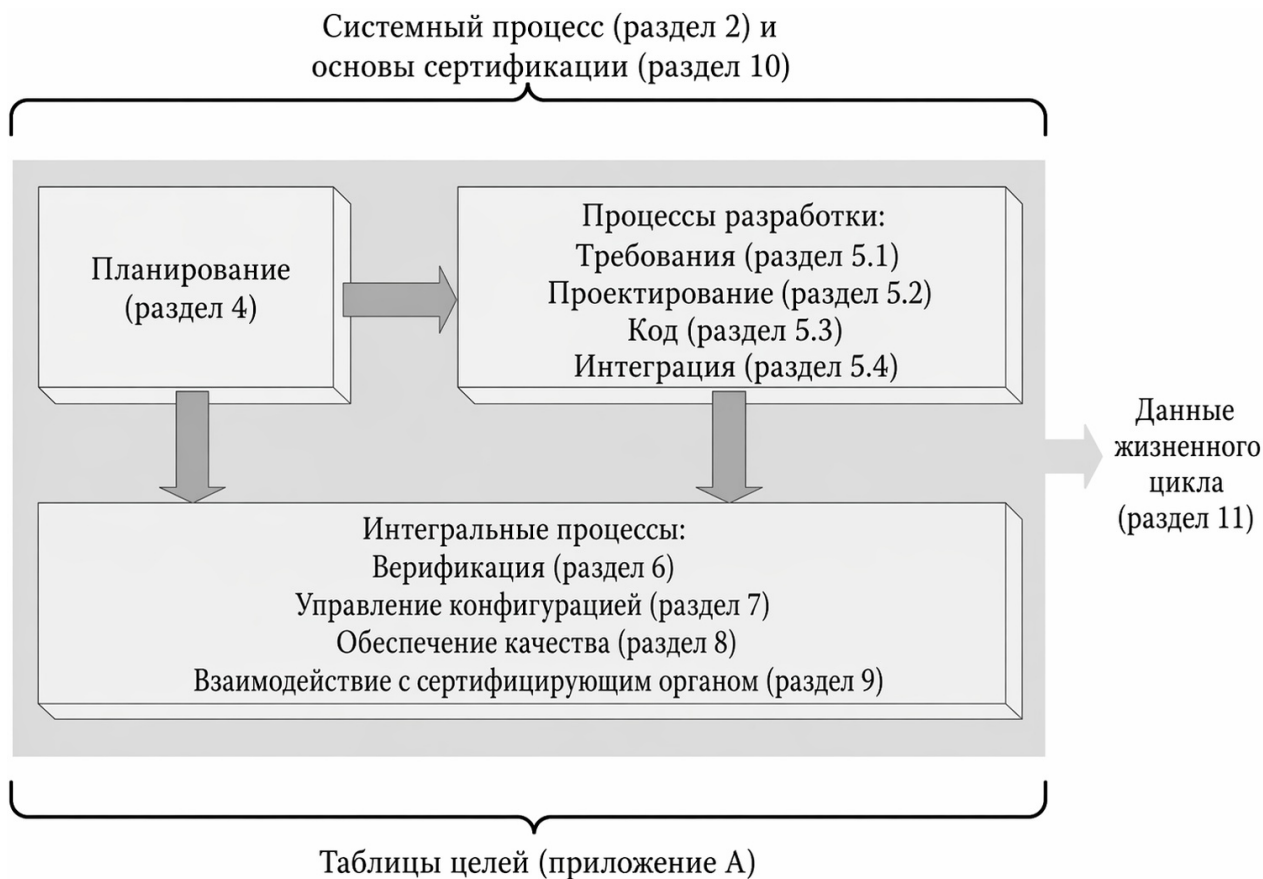


Рисунок 4.2 Обзор DO-178C.

- *DO-178C section 1* служит введением в документ.
- *DO-178C section 2* включает системную концептуальную основу, согласованную с ARP4754A.
- *DO-178C section 3* дает краткий обзор процессов жизненного цикла программного обеспечения по DO-178C: planning (планирование), development (разработка: требования, проектирование, код и интеграция), verification (верификация), software configuration management (управление конфигурацией ПО), software quality assurance (обеспечение качества ПО) и certification liaison (взаимодействие с сертифицирующим органом).
- *DO-178C section 4* объясняет цели и мероприятия процесса планирования. Цели планирования кратко сведены в таблице A-1 DO-178C.
- *DO-178C section 5* объясняет цели и мероприятия процессов разработки. Процессы разработки включают этапы требований, проектирования, кода и интеграции. Цели разработки кратко сведены в таблице A-2 DO-178C.
- *DO-178C section 6* описывает процесс верификации. Верификация является интегральным процессом, который начинается с верификации планов и продолжается вплоть до отчетности по результатам верификации и их рассмотрения. Верификация включает

reviews (рассмотрения), analyses (анализы) и tests (тесты). Она играет значительную роль в гарантии разработки ПО: более половины целей DO-178C являются целями верификации. Цели верификации кратко сведены в таблицах A-3-A-7 DO-178C.

- *DO-178C section 7* объясняет процесс управления конфигурацией ПО (SCM), который включает change control (управление изменениями) и problem reporting (регистрацию сообщений о проблемах (PR)). Цели управления конфигурацией кратко сведены в таблице A-8 DO-178C.
- *DO-178C section 8* дает руководство по процессу гарантии качества ПО (SQA). Цели процесса гарантии качества ПО кратко сведены в таблице A-9 DO-178C.
- *DO-178C sections 9 and 10* кратко излагают процесс взаимодействия с сертифицирующим органом и общий процесс сертификации воздушного судна или двигателя.
- Цели взаимодействия с сертифицирующим органом кратко сведены в таблице A-10 DO-178C.
- *DO-178C section 11* определяет данные жизненного цикла, формируемые при планировании, разработке и верификации программного обеспечения. Во многих случаях термин *software life cycle data* также называют *data items* или *artifacts*. Ожидаемое содержимое каждого элемента данных жизненного цикла кратко описано в разделе 11 DO-178C. Каждый элемент данных будет рассмотрен в последующих главах.
- *DO-178C section 12* содержит руководство по ряду дополнительных вопросов, выходящих за рамки предыдущих разделов, включая ранее разработанное ПО (PDS), tool qualification levels (уровни квалификации инструмента, TQL) и alternative methods (альтернативные методы), такие как exhaustive input testing (исчерпывающее тестирование входов), multiple-version dissimilar software (многоверсионное разнородное ПО), software reliability models (модели надежности программного обеспечения) и product service history (история эксплуатации изделия).
- *DO-178C Annex A* содержит таблицы по каждому процессу, которые соотносят цели и выходные данные. Таблицы Annex A DO-178C кратко сводят цели, применимость по уровню ПО, требования к независимости для конкретной цели, выходные данные, которые обычно формируются для выполнения цели, и объем управления конфигурацией, требуемый для этих выходных данных. В разделе 4.2.2 подробнее объясняется взаимосвязь между таблицами Annex A DO-178C и основной частью документа DO-178C, а также то, как интерпретировать эти таблицы.
- *DO-178C Annex B* содержит глоссарий терминов, используемых в DO-178C. DO-178C использует общепринятые определения из программной инженерии, но адаптирует

некоторые из них к специфике авиационной области.

- *DO-178C appendix A* содержит справочную информацию и техническое задание комитета. *DO-178C appendix B* содержит список членов комитета. Любой, кто участвовал как минимум в двух заседаниях, был включен в этот список, поэтому он получился довольно длинным.

4.2.1 Отличия DO-278A и DO-178C

DO-278A носит название *Guidelines for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance*. DO-278A ориентирован на наземное программное обеспечение CNS/ATM (связь, навигация, наблюдение / организация воздушного движения), которое может непосредственно влиять на безопасность воздушных судов. Основные отличия между DO-178C и DO-278A приведены ниже.

- В DO-178C используется термин *airborne software* (бортовое программное обеспечение), тогда как в DO-278A используется *CNS/ATM software* (ПО связи, навигации, наблюдения и управления воздушным движением).
- В DO-178C используется термин *certification* (сертификация), тогда как в DO-278A используется *approval* (одобрение).
- В DO-178C используются требования летной годности, тогда как в DO-278A используются применимые требования одобрения.
- В DO-178C используются файлы компонентов параметрических данных, тогда как в DO-278A используются файлы элементов адаптационных данных.
- В DO-178C используется План сертификации ПО (PSAC), тогда как в DO-278A используется План аспектов ПО для одобрения (PSAA).
- Раздел 2 DO-278A немного отличается от DO-178C, поскольку связь с процессом оценки безопасности для наземных систем CNS/ATM (связь, навигация, наблюдение / организация воздушного движения) отличается от таковой для воздушных судов. DO-278A использует уровни гарантии разработки 1-6 (AL1-AL6) вместо уровней ПО А-Е. Уровни гарантии разработки по DO-278A эквивалентны уровням DO-178C, за исключением того, что у DO-278A есть дополнительный уровень между уровнями С и D документа DO-178C. В таблице 4.3 показано соответствие между уровнями DO-278A и DO-178C.

Таблица 4.3 Соответствие уровней DO-178C и DO-278A

Уровень ПО по DO-178C	Уровень гарантии разработки по DO-278A
A	AL1
B	AL2

Уровень ПО по DO-178C	Уровень гарантии разработки по DO-278A
C	AL3
Нет прямого эквивалента	AL4
D	AL5
E	AL6

Источник: RTCA DO-278A, *Guidelines for Communications, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance*, RTCA, Inc., Вашингтон, округ Колумбия, декабрь 2011 года.

- В DO-278A имеются краткие разделы, посвященные security requirements (требованиям по защищенности), adaptability (адаптируемости), cutover (горячему переключению) и post-development life cycle (послеразработочному жизненному циклу), которых нет в DO-178C. Раздел DO-278A о service experience (опыте эксплуатации) немного отличается от раздела DO-178C об истории эксплуатации изделия.
- В DO-278A есть объемный раздел о готовом коммерческом ПО (COTS), которого нет в DO-178C.

4.2.2 Обзор таблиц целей DO-178C из Annex A

Часто говорят, что DO-178C лучше читать с конца к началу. Annex A DO-178C кратко сводит цели, применимые разделы и требуемые данные из разделов 4-11 DO-178C. Однако если не читать вводные разделы, таблицы целей Annex A DO-178C не дадут полного понимания предметной области. По сути, таблицы Annex A DO-178C предоставляют дорожную карту применения целей по уровням ПО. В этом подразделе кратко объясняется предмет каждой таблицы, взаимосвязь таблиц и способы их интерпретации. Цели будут дополнительно обсуждаться в главах 5-12 по мере более глубокого рассмотрения технического материала.

Таблица 4.4 содержит названия десяти таблиц Annex A DO-178C, а также число целей, включенных в каждую таблицу. Таблица 4.5 показывает количество целей, применимых для каждого уровня ПО. На рисунке 4.3 показана взаимосвязь десяти таблиц целей DO-178C. Процесс планирования управляет остальными процессами, а процессы управления конфигурацией, обеспечения качества и взаимодействия с сертифицирующим органом выполняются на протяжении всего проекта.

Таблица 4.4 Сводка таблиц Annex A документа DO-178C

Таблица Annex A	Число целей	Название таблицы Annex A DO-178C
A-1	7	Процесс планирования программного обеспечения

A-2	7	Процессы разработки ПО
A-3	7	Верификация выходных данных процесса требований к ПО
A-4	13	Верификация выходных данных процесса проектирования ПО
A-5	9	Верификация выходных данных процесса кодирования и интеграции ПО
A-6	5	Тестирование выходных данных процесса интеграции
A-7	9	Верификация результатов процесса верификации
A-8	6	Процесс управления конфигурацией ПО
A-9	5	Процесс гарантии качества ПО
A-10	3	Процесс взаимодействия с сертифицирующим органом

Источник: RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Вашингтон, округ Колумбия, декабрь 2011 года.

Таблица 4.5 Количество целей DO-178C по уровням ПО

Уровень ПО	A	B	C	D	E
Количество целей	71	69	62	26	0



Рисунок 4.3 Взаимосвязь таблиц Annex A документа DO-178C.

В таблице 4.6 показана характерная цель из DO-178C (таблица A-6, цель 3). В каждой таблице целей есть пять основных разделов. Каждый из этих разделов кратко описан ниже:

Таблица 4.6 Пример цели DO-178C

Элемент	Содержание
Цель	Исполняемый объектный код соответствует требованиям низкого уровня.
Ссылка на описание цели	6.4.с.
Связанные мероприятия	6.4.2, 6.4.2.1, 6.4.3 и 6.5.
Применимость по уровням ПО	Уровень А: применяется с независимостью. Уровень В: применяется с независимостью. Уровень С: применяется без требования независимости. Уровень D: не применяется.
Выходные данные	Тестовые примеры и процедуры верификации программного обеспечения, ссылка 11.13. Результаты верификации ПО, ссылка 11.14. Данные трассировки, ссылка 11.21.
Категория контроля по уровням ПО	Для тестовых примеров и процедур: КК1 для уровней А и В, КК2 для уровней С и D. Для результатов верификации: КК2 для уровней А, В, С и D. Для данных трассировки: КК1 для уровней А и В, КК2 для уровней С и D.

Источник: RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Вашингтон, округ Колумбия, декабрь 2011 года.

1. Раздел *Objective* (цель) кратко формулирует цель и дает ссылку назад на основную часть документа, где эта цель объясняется. Цели – это *что* необходимо сделать. Они не объясняют, *как* именно это делать. Для полного понимания цели важно прочитать раздел, на который дана ссылка. Большинство формулировок достаточно интуитивны, но некоторые из них более сложны. Цели будут разъясняться в главах 5-12, включая и более трудные для понимания.
2. Раздел *Activity* (мероприятие) дает ссылку назад на основную часть DO-178C, где приведены рекомендуемые мероприятия по выполнению цели. Эти мероприятия отражают типовой подход к выполнению цели, однако в планах могут быть предложены и иные подходы.
3. Раздел *Applicability by Software Level* (применимость по уровню ПО) определяет, какие цели применимы для каких уровней ПО. Как отмечалось ранее, все цели применимы к уровню А; однако уровни В-D представляют собой подмножество. Символ ○ или • в столбцах А, В, С или D означает, что данная цель должна быть выполнена для соответствующего уровня ПО. Закрашенный кружок (●) означает, что цель должна быть выполнена с независимостью. Для целей верификации независимость просто означает, что верификацию выполняет отдельный человек или отдельный инструмент, то есть тот, кто не создавал исходный верифицируемый артефакт. Независимость будет объяснена далее в этой книге (раздел 9.3 для независимости верификации и раздел 11.1.3 для независимости обеспечения качества программного обеспечения).

Всего в DO-178C 71 цель. Для уровня А необходимо выполнить все 71 цель. Для уровня В и ниже это число меньше, как показано в таблице 4.5. В зависимости от подхода к разработке программного обеспечения некоторые цели могут быть неприменимы (например,

если partitioning (обособление) не используется, то цель 13 таблицы А-4 не применяется, а если отсутствуют parameter data (параметрические данные), то цели 8 и 9 таблицы А-5 не применяются).

4. Столбцы *Output* (выходные данные) определяют данные, которые должны быть сформированы для доказательства выполнения цели, а также соответствующую ссылку на раздел 11 DO-178С, где дается краткое объяснение каждого элемента данных. В мире сертификации есть поговорка: “Если это не записано, значит, этого не было”. Столбец выходных данных указывает данные, используемые для доказательства соответствия целям. В таблице 4.7 приводится сводка всех элементов данных из раздела 11 DO-178С, на которые есть ссылки в таблицах Annex А. Элементы, помеченные звездочкой, всегда требуются, если применима хотя бы одна цель, ссылающаяся на соответствующий элемент данных.

Таблица 4.7 Обзор данных жизненного цикла программного обеспечения

Элемент данных	Назначение
*11.1 План сертификации ПО (PSAC – Plan for Software Aspects of Certification)	Верхнеуровневый план программного обеспечения, который фиксирует договоренности с сертифицирующим органом.
11.2 План разработки ПО (SDP – Software Development Plan)	Описывает процедуры разработки и жизненного цикла, направляющие команду разработки и обеспечивающие соответствие целям разработки DO-178С.
11.3 План верификации ПО (SVP – Software Verification Plan)	Описывает процедуры верификации, направляющие верификаторов и обеспечивающие соответствие целям верификации DO-178С.
11.4 План управления конфигурацией ПО (SCMP – Software Configuration Management Plan)	Устанавливает среду управления конфигурацией, процедуры, мероприятия и процессы, применяемые на протяжении разработки и верификации ПО.
11.5 План гарантии качества ПО (SQAP – Software Quality Assurance Plan)	Определяет, как обеспечение качества будет контролировать проект для подтверждения соответствия целям DO-178С, планам и стандартам.
11.6 Стандарты требований к ПО (Software Requirements Standards)	Задают рекомендации, методы, правила и инструменты для авторов требований.
11.7 Стандарты проектирования ПО (Software Design Standards)	Задают рекомендации, методы, правила и инструменты для проектировщиков.
11.8 Стандарты кодирования ПО (Software Code Standards)	Задают рекомендации, методы, правила и инструменты для эффективного использования языка программирования.
11.9 Данные требований высокого уровня к ПО (Software Requirements Data)	Определяет требования высокого уровня к ПО и производные требования высокого уровня.
11.10 Описание проекта ПО (SDD – Software Design Description)	Определяет программную архитектуру, требования низкого уровня к ПО и производные требования низкого уровня.
11.11 Исходный код (Source Code)	Набор файлов кода, который вместе с данными компиляции, компоновки и загрузки используется для создания исполняемого объектного кода и интеграции его в целевой вычислитель.
11.12 Исполняемый объектный код (Executable Object Code)	Код, который непосредственно считывается процессором целевого вычислителя.
11.13 Примеры и Процедуры верификации (SVCP – Software Verification Cases and Procedures)	Описывают, как реализуются процессы верификации ПО.
11.14 Результаты верификации ПО (SVR – Software Verification Results)	Представляет выходные данные процессов верификации.
11.15 Указатель конфигурации среды ЖЦ ПО (SECI – Software Life Cycle Environment Configuration Index)	Идентифицирует программную среду, включая все инструменты, используемые для разработки, управления, сборки, верификации и загрузки ПО.
*11.16 Указатель конфигурации ПО (SCI – Software Configuration Index)	Идентифицирует конфигурацию программного продукта, включая исходный код, исполняемый объектный код и сопутствующие данные ЖЦ ПО. Также включает инструкции по сборке и загрузке.
11.17 Сообщения о проблемах (PR – Problem Reports)	Идентифицируют проблемы продукта и процесса, чтобы обеспечить их разрешение.

Элемент данных	Назначение
11.18 Записи управления конфигурацией ПО (Software Configuration Management Records)	Включают результаты различных действий по управлению конфигурацией.
11.19 Записи гарантии качества ПО (Software Quality Assurance Records)	Включают результаты мероприятий по обеспечению качества, в том числе ревизии соответствия ПО.
*11.20 Итоговый отчет разработки ПО (SAS – Software Accomplishment Summary)	Суммирует соответствие DO-178C, любые отклонения от Плана сертификации ПО (PSAC), характеристики ПО и открытые сообщения о проблемах (PR).
11.21 Данные трассировки (Trace Data)	Предоставляют свидетельства трассировки между требованиями, проектом, кодом и верификационными данными.
11.22 Файл компонента параметрических данных (Parameter Data Item File)	Содержит данные, такие как конфигурационные данные, которые могут непосредственно использоваться целевым вычислителем.

Источник: RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Вашингтон, округ Колумбия, декабрь 2011 года.

Примечание: документы, отмеченные звездочкой в исходной таблице, как минимум передаются в сертифицирующий орган.

5. Раздел *Control Category by Software Level* (категория контроля по уровню ПО) указывает уровень управления конфигурацией, который применяется к соответствующему элементу выходных данных. Один символ в исходной таблице означает категорию контроля 1 (KK1, control category #1) и требует более строгого процесса управления конфигурацией. Другой символ означает категорию контроля 2 (KK2, control category #2) и является менее строгим. KK1 и KK2 объясняются в главе 11.

4.3 DO-330: Соображения по квалификации программных инструментов

DO-178C определяет программный инструмент так: “Компьютерная программа, используемая для помощи в разработке, тестировании, анализе, создании или модификации другой программы либо ее документации. Примерами являются автоматизированный инструмент проектирования, компилятор, инструменты тестирования и инструменты модификации” [1]. Из-за увеличившегося числа используемых инструментов и большого количества сторонних производителей инструментов руководство по квалификации инструментов было расширено и вынесено из основного текста DO-178C в отдельный документ DO-330. Раздел 12.2 DO-178C объясняет, как определить, нуждается ли инструмент в квалификации и до какого уровня его необходимо квалифицировать. Определено пять уровней TQL; TQL-1 является наиболее строгим. DO-330 объясняет, что нужно сделать для квалификации инструмента. DO-330 организован аналогично DO-178C и имеет собственный набор целей. DO-330 представляет собой самостоятельный документ, который позволяет разработчикам инструментов создавать инструменты, пригодные для квалификации, не обладая детальным знанием самого DO-178C. DO-330 был создан так, чтобы быть как можно более независимым от конкретной области применения, поэтому он может использоваться и в других областях (например, разработчиками аэронавигационных баз данных, разработчи-

ками электронной аппаратуры, разработчиками инструментов оценки системной безопасности, системными разработчиками). Руководство DO-330 применяется к инструментам, реализованным программно, то есть не распространяется на инструменты, реализованные аппаратно. Для каждой конкретной области, которая ссылается на это руководство, может потребоваться некоторая адаптация, чтобы определить соответствующие уровни TQL и использование инструмента в жизненном цикле данной области. Однако сам процесс квалификации в целом относительно независим от области, в которой используется инструмент. В приложениях C и D DO-330 приведен ряд часто задаваемых вопросов по квалификации инструментов. DO-330 и квалификация инструментов рассматриваются в главе 13.

4.4 Технологические дополнения к DO-178C

Чтобы разработать подход, способный изменяться вместе с технологиями и тенденциями, комитет RTCA и EUROCAE решил подготовить технологические дополнения. Замысел состоял в том, что по мере изменения технологий основная часть DO-178C сможет оставаться неизменной, а технологические дополнения можно будет изменять или добавлять по мере необходимости. Одновременно с DO-178C были опубликованы три технологических дополнения. Каждое технологическое дополнение основано на целях и расширяет цели основного текста DO-178C применительно к своей конкретной технологии. Каждое дополнение объясняет, как цели из основного текста DO-178C применяются, изменяются или заменяются. Ниже каждое дополнение кратко резюмируется; более подробно они будут рассмотрены в последующих главах. Если одновременно используются несколько дополнений, План сертификации ПО (PSAC) должен объяснять, какое дополнение и какие цели применимы к каждой части ЖЦ ПО и к формируемым данным.

4.4.1 DO-331: Дополнение по модельно-ориентированной разработке

DO-331, озаглавленный *Model-Based Development and Verification Supplement to DO-178C and DO-278A*, определяет модель следующим образом [3]:[*]

Абстрактное представление набора программных аспектов системы, используемое для поддержки процесса разработки программного обеспечения или процесса верификации ПО. Это дополнение [DO-331] рассматривает модель(и), обладающие следующими характеристиками:

- a. Модель полностью описана с использованием явно определенной нотации моделирования. Нотация моделирования может быть графической и/или текстовой.
- b. Модель содержит программные требования и/или определение архитектуры программного обеспечения.
- c. Модель имеет такую форму и такой тип, которые используются для непосредственного

анализа или поведенческой оценки в рамках процесса разработки программного обеспечения либо процесса верификации ПО.

DO-331 основан на основном тексте DO-178C и изменяет или добавляет к нему руководство. В дополнении DO-331 рассматриваются как *model-based development* (модельно-ориентированная разработка), так и *model-based verification* (модельно-ориентированная верификация). DO-331 содержит руководство для *specification models* (моделей спецификации) и *design models* (моделей проектирования), а также для анализа покрытия моделей и моделирования поведения. Большинство различий между DO-178C и DO-331 сосредоточено в разделах 5 и 6, то есть в процессах разработки и верификации. Имеются также влияния на планирование (раздел 4), данные ЖЦ ПО (раздел 11) и таблицы целей Annex A. Модельно-ориентированная разработка и верификация, а также DO-331, подробнее рассматриваются в главе 14.

4.4.2 DO-332: Дополнение по объектно-ориентированной технологии

DO-332, озаглавленный *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*, поясняет: “Object-oriented technology (объектно-ориентированная технология) – это парадигма системного анализа, проектирования, моделирования и программирования, сосредоточенная на объектах. В объектно-ориентированных языках есть концепции и приемы, которые необходимо учитывать при рассмотрении безопасности программного обеспечения” [4]. DO-332 выделяет наиболее существенные вопросы, связанные с объектно-ориентированной технологией (ООТ), и дает руководство о том, что необходимо сделать для их учета. Кроме того, DO-332 рассматривает шесть *related techniques* (связанных техник), обычно ассоциируемых с ООТ, хотя и не ограниченных ею: *parametric polymorphism* (параметрический полиморфизм), *overloading* (перегрузку), *type conversion* (преобразование типов), *exception management* (управление исключениями), *dynamic memory management* (управление динамической памятью) и *virtualization* (виртуализацию). Как будет объяснено в главе 15, руководство по этим *related techniques* может быть применимо и к проектам, не использующим объектно-ориентированную технологию (ООТ), если языки в них применяют такие продвинутые техники. Как и другие дополнения, DO-332 основан на основном тексте DO-178C и указывает, где руководство необходимо изменить или усилить. Некоторые изменения по отношению к DO-178C указаны в разделах 4-6 и 11 документа DO-332. Однако в целом эти изменения относительно невелики. Помимо руководства, содержащегося в разделах 1-12 DO-332, дополнение включает Annex D, в котором определяются *vulnerabilities* (уязвимости) объектно-ориентированной технологии и связанных техник (ООТ&РТ), а также дается руководство о том, что нужно сделать для устранения этих уязвимостей. Annex D с уязвимостями является уникальной особенностью DO-332; у других дополнений такого раздела нет. Как и DO-330 и другие дополнения,

DO-332 имеет приложение, в котором кратко сводятся часто задаваемые вопросы. FAQ по объектно-ориентированной технологии и связанным техникам (OOT&RT) разделены на следующие категории: (1) общие, (2) требования, (3) проектирование, (4) программирование и (5) верификация. DO-332 и объектно-ориентированная технология (OOT) рассматриваются подробнее в главе 15.

4.4.3 DO-333: Дополнение по формальным методам

DO-333, озаглавленный *Formal Methods Supplement to DO-178C and DO-278A*, определяет formal methods (формальные методы) так: “Описательные нотации и аналитические методы, используемые для построения, разработки и рассуждения о математических моделях поведения системы. Формальный метод – это формальный анализ, выполняемый над формальной моделью” [5]. DO-178B определял формальные методы как alternative method (альтернативный метод); однако соответствующее руководство было расплывчатым и не получило широкого применения. После публикации DO-178B в области формальных методов произошел прогресс, а инструменты и техники были улучшены, что сделало эту технологию более осуществимой. Поэтому раздел о формальных методах был исключен из DO-178C и вынесен в дополнение DO-333. DO-333 использует основной текст DO-178C в качестве основы и дополняет его руководством, специфичным для формальных методов. Основные разделы, в которых DO-178C и DO-333 различаются, это разделы 5, 6 и 11, а также таблицы целей Annex A. Большинство остальных разделов остается относительно неизменным. Более подробная информация о формальных методах и DO-333 приводится в главе 16.

4.5 DO-248C: Вспомогательный материал

DO-248C, озаглавленный *Supporting Information for DO-178C and DO-278A*, представляет собой сборник FAQ (часто задаваемых вопросов) и DP по темам DO-178C. Он заменяет DO-248B и очень похож на DO-248B, но имеет следующие отличия:

- Как можно заключить из названия, DO-248C касается DO-278A так же, как и DO-178C.[*]
- В DO-248C нет errata (перечня исправлений), поскольку на момент публикации DO-248C никаких исправлений для DO-178C выявлено не было.
- Некоторые FAQ и DP из DO-248B были удалены из DO-248C, поскольку соответствующая информация теперь существует в других местах (например, в DO-178C, DO-330, дополнениях или в системных документах и документах по безопасности [ARP4754A и ARP4761]).
- Некоторые FAQ и DP были обновлены для согласования с DO-178C (например, FAQ #42 о структурном покрытии на уровне объектного кода и FAQ #67 об анализе межком-

понентной связности по данным и по управлению).

- Были добавлены некоторые новые FAQ и DP для разъяснения актуальных тем. Часть новых FAQ или DP была добавлена для разъяснения новых тем DO-178C (например, DP #20 о parameter data items (параметрических элементах данных)). Другие FAQ или DP были добавлены для разъяснения тем, ставших более актуальными после публикации DO-248B (например, DP #16 об управлении кэшем, DP #17 об арифметике с плавающей точкой и DP #21 об одиночном радиационном сбое (single event upset)). Некоторые FAQ или DP были добавлены, чтобы охватить ряд распространенных тем, связанных с сертификацией (например, FAQ #81 об объединении требований высокого и низкого уровней, FAQ #83 о требованиях низкого уровня, представленных в форме псевдокода, и DP #19 о независимости).
- В DO-248C был добавлен раздел 5; он содержит обоснование для DO-178C, DO-278A, DO-330 и дополнений.

DO-248C не предназначен для чтения последовательно от начала до конца. Это, скорее, серия коротких материалов, объясняющих аспекты DO-178C, которые часто понимаются неправильно или вызывают трудности. Ключом к использованию DO-248C являются его приложения C и D. Приложение C документа DO-248C содержит указатель ключевых слов документа. Приложение D документа DO-248C содержит сопоставление материалов DO-248C с разделами DO-178C. При исследовании темы или раздела DO-178C можно использовать приложение C или D для поиска соответствующих разделов DO-248C. В электронной версии для повышения удобства использования имеются гиперссылки. Несколько FAQ и DP из DO-248C будут упоминаться по ходу этой книги как относящиеся к рассматриваемой теме.

Следует отметить, что FAQ и DP по квалификации инструментов или по технологическим дополнениям включены в соответствующие документы, а не в DO-248C.

Ожидается, что по мере накопления авиационным сообществом опыта применения DO-178C документ DO-248C будет обновляться, чтобы отражать errata и давать дополнительные разъяснения.

Литература

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
2. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
3. RTCA DO-331, *Model-Based Development and Verification Supplement to DO-178C and*

DO-278A (Washington, DC: RTCA, Inc., December 2011).

4. RTCA DO-332, *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
5. RTCA DO-333, *Formal Methods Supplement to DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
6. RTCA DO-278A, *Guidelines for Communications, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance* (Washington, DC: RTCA, Inc., December 2011).

*European Organization for Civil Aviation Equipment (Европейская организация по оборудованию для гражданской авиации).

*Совместным комитетом являются RTCA Special Committee #205 (SC-205) и EUROCAE Working Group #71 (WG-71). Комитет был разделен на исполнительный комитет и семь подгрупп.

*CAST – это группа международных сертифицирующих органов, стремящихся гармонизировать свои позиции по бортовому программному обеспечению и авиационной электронной аппаратуре в документах CAST.

*Скобки добавлены для пояснения.

*Далее в этом разделе всякий раз, когда упоминается DO-178C, то же самое относится и к DO-278A.

Глава 5. Планирование программного обеспечения

Сокращения

Сокращение	Расшифровка
KK1	Категория контроля 1 (control category 1)
KK2	Категория контроля 2 (control category 2)
COTS	Готовое коммерческое ПО (commercial off-the-shelf)
CRC	Циклический избыточный контроль (cyclic redundancy check)
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
MISRA	Ассоциация надежности программного обеспечения автомобильной промышленности (Motor Industry Software Reliability Association)
PQA	Гарантия качества продукта (product quality assurance)
PQE	Инженер по качеству продукта (product quality engineer)
PR	Сообщение о проблеме (problem report)
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
RTOS	ОСРВ (операционная система реального времени, real-time operating system)
SAS	Итоговый отчёт разработки ПО (Software Accomplishment Summary)
SCI	Указатель конфигурации ПО (Software Configuration Index)
SCM	Управление конфигурацией ПО (software configuration management)
SCMP	План управления конфигурацией ПО (Software Configuration Management Plan)
SDP	План разработки ПО (Software Development Plan)
SECI	Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index)
SOI	Этап взаимодействия с сертифицирующим органом (stage of involvement)
SQA	Гарантия качества ПО (software quality assurance)
SQAP	План гарантии качества ПО (Software Quality Assurance Plan)

Сокращение	Расшифровка
SVP	План верификации ПО (Software Verification Plan)
SVCP	Примеры и Процедуры верификации (Software Verification Cases and Procedures)

5.1 Введение

Вы, вероятно, слышали выражение: “Если вы не планируете, вы планируете провал”. Или, возможно, слышали другое, одно из моих любимых: “Если целиться в ничто, вы будете попадать в него каждый раз”. Хорошее программное обеспечение не возникает само собой – для него требуется обширное планирование, а также хорошо подготовленный персонал, способный выполнить план и адаптировать его.

Процесс планирования по DO-178C (КТ-178С) включает разработку планов и стандартов, специфичных для проекта. Соответствие DO-178C (и гарантия разработки в целом) требует, чтобы планы были и документированы, и соблюдались. Если планы написаны так, чтобы охватывать применимые цели DO-178C, то их выполнение обеспечит соответствие этим целям. Следовательно, соответствие DO-178C и одобрение со стороны сертифицирующего органа зависят от успешной работы по планированию.

DO-178C определяет пять планов и три стандарта и поясняет ожидаемое содержание каждого документа. Хотя большинство организаций следуют рекомендации “пять планов и три стандарта”, допустимо упаковывать планы и стандарты так, как это наилучшим образом соответствует потребностям организации. Хотя это и не рекомендуется, разработчик может объединить все планы и стандарты в одном документе (имейте в виду, что в этом случае весь документ потребуются представить сертифицирующему органу). Использование нетрадиционной структуры планирования может создать некоторые трудности при первом взаимодействии с сертифицирующим органом и может потребовать дополнительных представлений; однако это допустимо, если все необходимые темы, указанные в DO-178C, обсуждены в достаточной степени. Независимо от того, как упакованы планы и стандарты, необходимо охватить рекомендуемое содержание из разделов 11.1-11.8 DO-178C, и документы должны быть согласованными.

5.2 Общие рекомендации по планированию

Прежде чем рассматривать каждый из пяти планов, давайте рассмотрим несколько общих рекомендаций по планированию.

Рекомендация 1: Убедитесь, что планы охватывают все применимые цели DO-178C и все применимые цели дополнений.[]* Планы должны быть написаны так, чтобы команда при

их выполнении соблюдала все применимые цели. Чтобы это обеспечить, полезно иметь отображение между целями DO-178C (и применимых дополнений) и планами. После завершения такое отображение часто включается в приложение к Плану сертификации ПО (PSAC).[†] Я рекомендую представлять это отображение в виде таблицы из четырех столбцов; каждый столбец описан ниже:

1. *Таблица/цель №*: указывает таблицу Annex A и номер цели из DO-178C (и применимых дополнений).
2. *Краткое описание цели*: включает краткое описание цели в том виде, в каком оно приведено в Annex A DO-178C и/или в применимых дополнениях.
3. *Ссылка на План сертификации ПО (PSAC)*: указывает разделы Плана сертификации ПО (PSAC), которые объясняют, как данная цель будет удовлетворена.
4. *Ссылка на другие планы*: указывает разделы планов команды (например, Плана разработки ПО, Плана верификации ПО, Плана управления конфигурацией ПО и Плана гарантии качества ПО), которые объясняют подробные мероприятия по удовлетворению цели.

Рекомендация 2: Пишите планы и стандарты таким образом, чтобы команды, реализующие их, могли им следовать. Целевая аудитория Плана сертификации ПО (PSAC) – это сертифицирующий орган; однако предполагаемой аудиторией других планов и стандартов являются команды (например, команда разработки, команда верификации и команда обеспечения качества), которые будут эти планы выполнять. Они должны быть написаны на таком уровне, чтобы участники команд могли их понять и должным образом выполнять. Если команда опытная, в планах может не требоваться столько деталей. Однако для менее опытной команды или для проекта с широким использованием аутсорсинга обычно требуются более подробные планы.

Рекомендация 3: Указывайте что, как, когда и кто. В каждом плане поясняйте, что будет делаться, как это будет делаться, когда это будет делаться (не конкретную дату, а момент в общей последовательности мероприятий) и кто будет это делать (не обязательно конкретные имена инженеров, а команды, которые будут выполнять задачи).

Рекомендация 4: Завершите планы и передайте их под управление конфигурацией до начала разработки программного обеспечения. Если планы не завершены на раннем этапе проекта, становится трудно задать предполагаемое направление, найти время, чтобы написать их позднее, и убедить сертифицирующий орган, что им действительно следовали. Даже если планы не выпущены до начала разработки, они должны быть хотя бы подготовлены в черновом виде и переданы под управление конфигурацией. Формализация (то есть

рассмотрение и выпуск) должна произойти как можно раньше. Любые изменения между черновыми версиями и выпущенными версиями планов должны быть доведены до команд, чтобы они соответствующим образом обновили данные и практики.

Рекомендация 5: Обеспечьте внутреннюю согласованность каждого плана и согласованность между планами. Это кажется здравым смыслом, но несогласованность – одна из самых частых проблем, которые я обнаруживаю при рассмотрении планов. Когда планы несогласованы, усилия одной команды могут подрывать усилия другой. Если План разработки ПО (SDP) говорит одно, а План верификации ПО (SVP) – другое, команды могут начать самостоятельно решать, чему следовать. Существует несколько способов обеспечить согласованность между планами, включая использование общих авторов и рецензентов. Один из самых эффективных способов, который я нашла для создания согласованных планов, – собрать технических руководителей и совместно потратить некоторое время на определение жизненного цикла программного обеспечения от начала до конца, включая мероприятия, входные критерии, выходные критерии и результаты каждой фазы. Большая белая доска или проекция с компьютера для фиксации общего видения подходят отлично. Когда жизненный цикл задокументирован и согласован, тогда можно писать планы.

Рекомендация 6: Определяйте согласованные критерии перехода между процессами во всех планах. Следует тщательно отнестись к определению согласованных критериев перехода между процессами во всех планах, особенно в планах разработки и верификации.

Рекомендация 7: Если разрабатывается несколько программных продуктов, могут быть полезны корпоративные шаблоны планов. Эти шаблоны могут дать отправную точку для планов, специфичных для проекта. Лучше всего создавать шаблоны на основе набора планов, использованных хотя бы в одном успешном проекте и учитывающих извлеченные уроки. Я рекомендую регулярно обновлять шаблоны на основе извлеченных уроков и обратной связи от сертифицирующих органов, заказчиков, уполномоченных представителей, других команд и т.д.

Рекомендация 8: Привлекайте экспертов по сертификации (например, уполномоченных представителей[]) как можно раньше в процессе планирования.* Получение мнения от уполномоченного представителя или человека с опытом сертификации может сэкономить время и предотвратить ненужные проблемы позднее в проекте.

Рекомендация 9: Добивайтесь поддержки планов со стороны руководства. Эффективное выполнение проекта зависит от понимания и поддержки планов со стороны руководства. Существует много способов добиться такой поддержки; один из самых эффективных – вовлекать руководство в процесс планирования в роли авторов или рецензентов. После того как планы написаны, руководству потребуется установить подробную стратегию их

реализации (например, определить необходимые ресурсы); успех этой стратегии зависит от их понимания планов.

Рекомендация 10: Включайте в планы надлежащий уровень детализации. Вместо того чтобы включать подробные процедуры в сами планы, процедуры могут находиться в инженерном руководстве или в рабочих инструкциях. Планы должны указывать эти процедуры и давать их краткое изложение, но сами детали не обязаны находиться непосредственно в планах. Например, анализ структурного покрытия и анализ связей по данным и по управлению обычно требуют определенных процедур и инструкций; эти процедуры могут быть оформлены отдельно от Плана верификации ПО (SVP). Такой подход дает определенную гибкость для выбора наилучшего решения тогда, когда придет время фактически выполнять План верификации ПО (SVP). Важно, чтобы планы ясно объясняли, какие процедуры применяются, как эти процедуры будут находиться под управлением конфигурацией, как будут происходить изменения и как любые изменения будут доводиться до команды.

Рекомендация 11: Проведите инструктаж для сертифицирующего органа до представления Плана сертификации ПО (PSAC). Помимо представления Плана сертификации ПО (PSAC) и, возможно, других планов сертифицирующему органу, полезно дать этому органу презентацию по запланированному подходу. Это особенно рекомендуется для новых или необычных продуктов, а также для компаний, у которых может не быть крепких рабочих отношений с местным сертифицирующим органом. Довольно часто устное обсуждение позволяет выявить вопросы, которые нужно решить до представления планов сертифицирующему органу.

Рекомендация 12: Распространяйте планы среди команд. Когда планы достигли достаточной зрелости, убедитесь, что команды понимают их содержание. Меня поражает, как много проектов я проверяю, где инженеры не знают, где найти планы или что в них сказано. Поскольку инженеры не всегда в восторге от чтения планов, это может потребовать больше, чем просто разослать электронное письмо со ссылкой. Обучение, презентации, обсуждения на стартовых совещаниях и специальные рассмотрения помогают убедиться, что команды понимают планы.

5.3 Пять программных планов

DO-178C определяет следующие пять планов:

1. План сертификации ПО.
2. План разработки ПО.
3. План верификации ПО.
4. План управления конфигурацией ПО.
5. План гарантии качества ПО.

Эти планы описываются в следующих подразделах.

5.3.1 План сертификации ПО (PSAC)

План сертификации ПО (PSAC) – это единственный план, который всегда представляется сертифицирующему органу. Он похож на договор между заявителем и сертифицирующим органом; поэтому чем раньше он будет подготовлен и согласован, тем лучше. План сертификации ПО (PSAC), который не представляют до позднего этапа проекта, вносит в проект риск. Этот риск состоит в том, что команда может дойти до конца проекта и лишь тогда обнаружить, что процессы и/или данные не соответствуют требованиям. Это может привести к срыву сроков и бюджета, значительной переделке и усиленному вниманию со стороны сертифицирующего органа.

План сертификации ПО (PSAC) дает высокоуровневое описание проекта в целом и объясняет, как будут удовлетворяться цели DO-178C (и применимых дополнений). Он также дает сводку по остальным четырем планам. Поскольку План сертификации ПО (PSAC) нередко является единственным планом, представляемым сертифицирующему органу, он должен быть самодостаточным. Ему не нужно повторять все, что сказано в планах по разработке, верификации, управлению конфигурацией и обеспечению качества, но он должен давать точную и согласованную сводку этих планов. Время от времени мне попадается План сертификации ПО (PSAC), который лишь указывает на другие документы (я называю его *PSAC-указателем*). Вместо того чтобы суммировать процессы разработки, верификации, управления конфигурацией и гарантии качества ПО, он просто ссылается на другие планы. Это, конечно, уменьшает объем повторяющегося текста. Однако это также означает, что другие планы, скорее всего, придется представлять сертифицирующему органу, чтобы он мог в полной мере понять процессы. В идеале План сертификации ПО (PSAC) должен содержать достаточно деталей, чтобы сертифицирующий орган понимал процессы и мог сделать корректное суждение о соответствии, но не настолько много, чтобы он просто повторял содержание других планов.

План сертификации ПО (PSAC) следует писать ясно и лаконично. Мне приходилось рассматривать Планы сертификации ПО (PSAC), в которых одно и то же повторялось по несколько раз и даже копировались разделы из DO-178B. Чем яснее документ, тем меньше в нем путаницы и тем выше вероятность, что он будет одобрен сертифицирующим органом.

Раздел 11.1 DO-178C дает сводку ожидаемого содержания Плана сертификации ПО (PSAC) и часто используется как основа его структуры. Ниже приведены типовые разделы Плана сертификации ПО (PSAC) вместе с кратким изложением содержания каждого раздела [1]:

- *Обзор системы*: этот раздел объясняет систему в целом и то, как программное обеспечение вписывается в систему. Обычно он занимает пару страниц.

- *Обзор ПО*: этот раздел объясняет предполагаемую функциональность ПО, а также любые архитектурные вопросы. Он тоже обычно состоит из пары страниц. Поскольку это план по ПО, важно объяснить именно то ПО, которое будет разрабатываться, а не только систему в целом.
- *Соображения по сертификации*: этот раздел объясняет способ подтверждения соответствия. Если используются дополнения к DO-178C, это хорошее место, чтобы объяснить, какое дополнение применяется к какой части ПО. Кроме того, в этом разделе обычно суммируется оценка безопасности, обосновывающая назначенные уровни ПО. Даже для ПО уровня А важно подробно объяснить, что именно определяет такое решение, и требуется ли какая-либо дополнительная архитектурная мера снижения риска. Многие проекты также описывают в этом разделе свои планы по поддержке аудитов этапов взаимодействия с сертифицирующим органом (SOI). Аудиты SOI рассматриваются в главе 12.
- *Жизненный цикл ПО*: этот раздел описывает фазы разработки ПО и интегральные процессы и обычно представляет собой сводку остальных планов (Плана разработки ПО, Плана верификации ПО, Плана управления конфигурацией ПО и Плана гарантии качества ПО).
- *Данные жизненного цикла ПО*: этот раздел перечисляет данные жизненного цикла, которые будут разрабатываться в проекте. Номера документов обычно еще имеют значения TBD или XXX, хотя иногда отдельные номера уже назначены. Часто этот раздел включает таблицу с 22 элементами данных из раздела 11 DO-178C, где приведены названия и номера документов. Обычной практикой также является указание, какие данные будут представляться сертифицирующему органу, а какие лишь предоставляться по запросу. Некоторые заявители указывают, будут ли данные относиться к категории контроля 1 (КК1) или категории контроля 2 (КК2), а также будут ли они утверждаться или рекомендоваться к утверждению уполномоченными представителями. В качестве альтернативы сведения о КК1/КК2 могут быть вынесены в План управления конфигурацией ПО (SCMP). Однако если используется общеорганизационный План управления конфигурацией ПО (SCMP), именно План сертификации ПО (PSAC) может содержать проектно-специфические сведения по управлению конфигурацией. КК1 и КК2 рассматриваются в главе 10 этой книги.
- *График*: этот раздел включает график разработки и одобрения ПО. Я еще ни разу не видела графика, который был бы выполнен точно; поэтому часто возникает вопрос: “Зачем вообще нужен график, если никто его не соблюдает?” Его цель в том, чтобы помочь и проекту, и сертифицирующему органу планировать свои ресурсы. Любые изменения графика в ходе проекта следует координировать с утверждающим органом.

Это не требует обновления Плана сертификации ПО (PSAC) само по себе, но если План сертификации ПО (PSAC) обновляется по другим причинам, график тоже следует обновить. Обычно график в Плане сертификации ПО (PSAC) достаточно высокоуровневый и включает основные программные вехи, например когда будут выпущены планы, завершены требования, завершено проектирование, завершен код, написаны и рассмотрены тестовые примеры, выполнено тестирование и подготовлен и представлен Итоговый отчёт разработки ПО (SAS).

- *Дополнительные соображения:* это один из самых важных разделов Плана сертификации ПО (PSAC), потому что он сообщает о любых особых обстоятельствах, о которых сертифицирующий орган должен знать. В сертификации важно всеми силами *избегать сюрпризов*. Ясное и лаконичное документирование всех дополнительных соображений помогает свести сюрпризы к минимуму и получить согласие сертифицирующего органа. DO-178C содержит неполный список таких тем, включая ранее разработанное ПО, готовое коммерческое ПО (COTS) и квалификацию инструментов. Если в вашем проекте есть что-либо необычное, это именно то место, где об этом следует сказать. Сюда могут относиться использование разделяемой ОСРВ (RTOS), офшорной команды или автоматизации. Кроме того, следует объяснить любой планируемый отключенный код или ПО с выбираемыми опциями. Хотя DO-178C этого не требует, я рекомендую в разделе дополнительных соображений Плана сертификации ПО (PSAC) приводить перечень всех инструментов проекта с кратким описанием того, как они используются, и обоснованием, почему инструмент нуждается или не нуждается в квалификации. Раскрытие такой информации уже на этапе планирования помогает предотвратить неприятные поздние открытия, особенно если какой-то инструмент на самом деле должен быть квалифицирован, но не был заранее распознан как требующий квалификации.

Как отмечалось ранее в рекомендации 1, полезно включать в приложение к Плану сертификации ПО (PSAC) перечень всех применимых целей DO-178C (и применимых дополнений) вместе с кратким объяснением того, как каждая цель будет удовлетворяться и где именно в планах она рассматривается. Уполномоченные представители и сертифицирующий орган ценят такую информацию, потому что она показывает, что проект действительно тщательно продумал детали соответствия DO-178C. И, к слову, если вы решите включить такое отображение целей в План сертификации ПО (PSAC), пожалуйста, убедитесь, что оно точное. Раз уж уполномоченные представители и сертифицирующие органы любят этот материал, они обычно его читают; а его точность и полнота помогают укрепить их доверие.

5.3.2 План разработки ПО (Software Development Plan, SDP)

План разработки ПО объясняет разработку программного обеспечения, включая фазы требований, проектирования, кодирования и интеграции (таблица А-2 DO-178C). Кроме того,

этот план часто кратко описывает и верификацию требований, проектирования, кода и интеграции (таблицы А-3 – А-5 DO-178С). Он пишется для разработчиков, которые будут разрабатывать требования, проектные данные и код, а также выполнять интеграцию. План должен быть написан так, чтобы направлять разработчиков к успешной реализации. Это означает, что он должен быть достаточно подробным, чтобы задавать ясное направление, но не настолько подробным, чтобы лишать инженеров возможности использовать инженерное суждение. Баланс тут тонкий. Как отмечалось в рекомендации 10, могут существовать более подробные процедуры или рабочие инструкции. Если это так, план должен ясно объяснять, какие процедуры применяются и когда именно они применяются, то есть он должен ссылаться на процедуры, а не включать их детали в сам документ. Иногда для подробных процедур требуется большая гибкость, например если они продолжают разрабатываться уже после выпуска планов. В таком случае план должен объяснять процесс разработки и контроля этих процедур; однако с таким подходом нужно быть осторожным, потому что иначе трудно гарантировать, что все инженеры пользуются правильной версией процедур.

Раздел 11.2 DO-178С определяет предпочтительное содержание Плана разработки ПО. Этот план включает описание трех крупных элементов: (1) стандартов, используемых при разработке (иногда сами стандарты даже включаются в план), (2) жизненного цикла программного обеспечения с объяснением каждой фазы и критериев перехода между фазами и (3) среды разработки, то есть методов и инструментов для требований, проектирования и кода, а также предполагаемых компилятора, компоновщика, загрузчика и аппаратных платформ. Ниже каждый из этих элементов поясняется отдельно:

1. *Стандарты*: каждый проект должен определить стандарты для требований, проектирования и кода. Эти стандарты задают правила и рекомендации, помогающие разработчикам создавать эффективные требования, проектные данные и код. Стандарты также задают ограничения, помогающие избежать ловушек, которые могут отрицательно сказаться на безопасности или функциональности программного обеспечения. Стандарты должны соответствовать используемой методологии или языку. Обычно План разработки ПО ссылается на стандарты, но в некоторых случаях стандарты могут быть включены прямо в этот план. Такое иногда происходит в компаниях, которые мало занимаются программной разработкой, или на небольших проектах. Три стандарта разработки рассматриваются далее в этой главе.
2. *Жизненный цикл программного обеспечения*: План разработки ПО определяет предполагаемый жизненный цикл разработки ПО. Обычно он основан на определенной модели жизненного цикла. В дополнение к указанию названия модели рекомендуется кратко объяснить саму модель, поскольку одна и та же формулировка не для всех означает одно и то же. Полезно включить графическое представление жизненного цикла и данных,

формируемых на каждой фазе. Среди моделей жизненного цикла, успешно использовавшихся в сертификационных проектах, встречаются каскадная модель, итеративная каскадная модель, быстрое прототипирование, спиральная модель и обратная разработка. Я настоятельно рекомендую избегать моделей *big bang*, *tornado* и *smoke-and-mirrors*.[*]

К сожалению, некоторые проекты указывают в своих планах одну модель жизненного цикла, а фактически следуют другой. Например, проекты иногда заявляют, что используют каскадную модель, потому что считают, будто этого требует DO-178C и именно этого предпочитает сертифицирующий орган. Однако DO-178C не требует каскадной модели, а заявлять ее без фактического применения – это отличный способ создать себе проблемы. Важно указать именно ту модель жизненного цикла, которую вы действительно собираетесь использовать, убедиться, что она удовлетворяет целям DO-178C, и затем реально следовать задокументированной модели. Если документированная модель жизненного цикла перестала соответствовать потребностям проекта, планы следует соответствующим образом обновить, если только с сертифицирующим органом не согласовано иное.

Как уже отмечалось ранее, План разработки ПО документирует критерии перехода для разработки программного обеспечения. Это включает входные и выходные критерии для каждой фазы разработки. Есть много способов документировать критерии перехода. Простым и эффективным способом может быть таблица. В ней перечисляются каждая фаза, мероприятия, выполняемые в этой фазе, критерии входа в фазу и критерии выхода из нее. В приложении А приведен пример таблицы критериев перехода для работ по разработке. Важно помнить, что DO-178C не задает обязательного порядка для мероприятий разработки, однако работы по верификации должны строиться сверху вниз, то есть требования нужно верифицировать до проектирования, а проектирование – до верификации кода.

3. *Среда разработки программного обеспечения*: помимо указания стандартов и описания жизненного цикла ПО План разработки ПО определяет и среду разработки. Это включает инструменты, используемые для разработки требований, проектных данных, исходного кода и исполнимого объектного кода, например компилятор, компоновщик, редактор, загрузчик и аппаратную платформу. Если полный перечень инструментов уже приведен в Плане сертификации ПО, как рекомендовалось ранее, План разработки ПО может просто сослаться на него. Однако План разработки ПО нередко идет дальше и более подробно объясняет, как именно эти инструменты используются в разработке ПО. Часто в Плане сертификации ПО и Плане разработки ПО не приводятся обозначения инструментов, потому что эта информация содержится в Указателе конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index, SECI). Чтобы избежать избыточности, некоторые проекты создают этот указатель уже на ранней стадии

и затем ссылаются на него из Плана разработки ПО. Если используется такой подход, начальная версия указателя должна быть готова одновременно с планами, даже если позже, по мере взросления проекта, ее почти наверняка придется обновлять.

Идентификация среды дает средство держать ее под контролем и обеспечивать воспроизводимость программного обеспечения. Неконтролируемые инструменты способны породить проблемы на этапах реализации, интеграции и верификации. Я лично видела ситуацию, когда разные разработчики использовали разные версии компилятора и разные настройки. Когда они начали интегрировать свои модули, стало, скажем так, заметно интереснее.

Если какие-либо инструменты разработки требуют квалификации, это должно быть объяснено в Плане разработки ПО. План сертификации ПО, возможно, уже содержит краткую сводку по квалификации, но План разработки ПО должен объяснять, как инструмент используется в общем жизненном цикле и откуда пользователи инструмента узнают о правильном порядке его применения, например через ссылку на требования к эксплуатации инструмента (Tool Operational Requirements) или руководство пользователя (User's Guide). Информация о квалификации инструментов приведена в главе 13.

В главах 6-8 приведена дополнительная информация соответственно о требованиях к программному обеспечению, проектировании и реализации.

5.3.3 План верификации ПО (Software Verification Plan, SVP)

Основная аудитория Плана верификации ПО – участники команды, которые будут выполнять работы по верификации, включая тесты. План верификации ПО тесно связан с Планом разработки ПО, поскольку усилия по верификации включают оценку данных, созданных на этапах разработки. Как уже отмечалось ранее, План разработки ПО часто дает высокоуровневую сводку по верификации требований, проектирования, кода и интеграции, например по коллегиальным рассмотрениям. План верификации ПО обычно дает больше деталей по рассмотрениям, включая подробности процесса рассмотрения, контрольные перечни, обязательных участников и так далее. Допустимо включать детали рассмотрений в План разработки ПО, а План верификации ПО использовать главным образом для тестов и анализов.

Независимо от того, как именно упакованы документы, должно быть ясно, какой план покрывает какие мероприятия.

Из всех планов именно План верификации ПО обычно сильнее всего меняется в зависимости от уровня программного обеспечения. Причина в том, что большинство различий между уровнями DO-178C сосредоточено именно в целях верификации. Обычно этот план объясняет, как команда будет удовлетворять целям из таблиц A-3 – A-7 DO-178C.

План верификации ПО объясняет организацию и состав команды верификации, а также то,

как выполняется требуемая DO-178C независимость. Хотя это и не обязательно, в большинстве проектов существует отдельная команда верификации, которая занимается разработкой и выполнением испытаний. DO-178C определяет несколько целей верификации, для которых требуется независимость, они отмечены закрашенными кружками в таблицах приложения A DO-178C. Независимость в смысле DO-178C не требует отдельной организации, но требует, чтобы один или несколько человек, либо, возможно, инструмент, не разрабатывавшие проверяемые данные, выполняли верификацию этих данных. По сути, независимость означает, что для проверки корректности, полноты, соответствия стандартам и тому подобного используется другая пара глаз и другой мозг, иногда еще и с инструментом в придачу. Подробнее о независимости верификации рассказывается в главе 9.

Верификация по DO-178C включает рассмотрения, анализы и тесты. План верификации ПО объясняет, как будут выполняться рассмотрения, анализы и тесты. Любые контрольные перечни, сопровождающие верификацию, либо включаются в этот план, либо приводятся в нем по ссылке.

Многие цели DO-178C могут удовлетворяться за счет рассмотрений. Таблицы A-3, A-4 и большая часть A-5 обычно закрываются через процесс коллегиального рассмотрения, который подробнее рассматривается в главе 6. Кроме того, некоторые цели таблицы A-7, например цели 1 и 2, тоже удовлетворяются рассмотрением. План верификации ПО объясняет процесс рассмотрения, включая сами процедуры рассмотрения или ссылки на них, критерии перехода для рассмотрений, а также контрольные перечни и записи, которыми оформляются рассмотрения. Обычно либо План верификации ПО, либо стандарты включают, или хотя бы ссылаются на, контрольные перечни для рассмотрения требований, проектных данных и кода. Эти списки помогают инженерам не пропускать важные критерии во время рассмотрения. Наиболее эффективными обычно оказываются короткие контрольные перечни; если они слишком подробны, ими, как правило, пользуются уже не полностью. Чтобы получить краткий, но полный контрольный перечень, я рекомендую разделять собственно пункты списка и подробные указания по ним на отдельные столбцы. Столбец контрольного перечня остается коротким, а столбец указаний содержит детальную информацию, чтобы рецензенты правильно понимали замысел каждого пункта. Такой подход особенно полезен для больших команд, команд с новыми инженерами и команд, использующих аутсорсинг. Он помогает задать планку рассмотрений и обеспечить согласованность. Контрольные перечни обычно включают пункты, помогающие убедиться, что оцениваются требуемые цели DO-178C, включая трассируемость, точность и согласованность, и что соблюдаются стандарты, но ими содержание не ограничивается.

Таблица A-6 DO-178C обычно удовлетворяется за счет разработки и выполнения тестов. План верификации ПО объясняет подход к тестам: то, как будут разрабатываться штатные

тесты и тесты на робастность; какая среда будет использоваться для их выполнения; как будет обеспечиваться трассируемость между требованиями, верификационными примерами и верификационными процедурами; как будут храниться результаты верификации; как будут определяться критерии прохождения или непрохождения; и где будут документироваться результаты тестов. Во многих случаях План верификации ПО ссылается на документ “Примеры и процедуры верификации” (Software Verification Cases and Procedures, SVCP), который подробно описывает план тестов, конкретные тестовые примеры и процедуры, испытательное оборудование, конфигурацию стенда и так далее.

Таблицы A-5 DO-178C, цели 6 и 7, и A-7, цели 3-8, обычно удовлетворяются, по крайней мере частично, выполнением анализов. Каждый запланированный анализ должен быть объяснен в Плане верификации ПО. Типичные анализы включают анализы трассируемости, то есть обеспечение полной и точной двунаправленной трассируемости между системными требованиями, требованиями высокого уровня к ПО, требованиями низкого уровня к ПО и тестовыми данными, анализ времени выполнения в худшем случае, анализ использования стека, анализ линковки, анализ загрузки, анализ карты памяти, анализ структурного покрытия и анализ покрытия требований. Эти анализы рассматриваются в главе 9. План верификации ПО (SVP) должен указывать предполагаемый подход для каждого анализа и место, где будут документироваться процедуры и результаты.

Поскольку при верификации часто используются инструменты, План верификации ПО перечисляет эти инструменты. Если список в Плане сертификации ПО обширен, как рекомендовалось ранее, можно просто сослаться на План сертификации ПО вместо повторения того же перечня. Однако План верификации ПО обычно дает больше подробностей о том, как именно каждый инструмент будет использоваться при верификации программного обеспечения, и ссылается на инструкции, необходимые для правильного применения инструментов. Как и в случае с инструментами разработки, План верификации ПО может ссылаться на указатель конфигурации среды ЖЦ ПО, чтобы указать сведения об инструментах, например версии и обозначения, вместо включения этих данных в сам план. Если используется такой подход, этот указатель должен быть завершен одновременно с планами и может потребовать ревизии до начала формального процесса верификации. Указатель конфигурации среды ЖЦ ПО рассматривается в главе 10.

Если для верификации программного обеспечения будет использоваться эмулятор или симулятор, это использование должно быть объяснено и обосновано в Плане верификации ПО. В некоторых случаях эмулятор или симулятор может потребовать квалификации. Эмуляция и моделирование рассматриваются в главе 9.

План верификации ПО также определяет критерии перехода для мероприятий по верифи-

кации. В приложении А приведен пример входных критериев, мероприятий и выходных критериев для верификационных работ проекта.

Если разрабатываемое и верифицируемое программное обеспечение содержит обособление, План верификации ПО должен объяснять, как будет верифицироваться целостность обособления, цель 13 таблицы А-4 DO-178С. Обособление рассматривается в главе 21.

Раздел 11.3 DO-178С также указывает, что План верификации ПО должен обсуждать допущения относительно корректности компилятора, компоновщика и загрузчика. Если используется оптимизация компилятора, это должно быть отражено в планах, поскольку она может повлиять на возможность получить анализ структурного покрытия или выполнить анализ соответствия исходного и объектного кода. Кроме того, План верификации ПО должен объяснять, как будет верифицироваться точность работы компоновщика. Если загрузчик используется без проверки целостности, например без циклического избыточного контроля (cyclic redundancy check, CRC), тогда придется верифицировать функциональность загрузчика. Если проверка целостности используется, План верификации ПО должен объяснить подход и обосновать достаточность этой проверки, например математически оценить точность алгоритма и убедиться, что этот циклический избыточный контроль достаточен для защищаемых данных.

Наконец, План верификации ПО должен объяснять, как будет выполняться повторная верификация. Если в ходе разработки вносятся изменения, будет ли повторно испытано вообще все или будет выполнен регрессионный анализ и заново проверены только измененные и затронутые элементы? Этот план должен объяснить планируемый подход, критерии, которые будут использоваться, и место, где будут документироваться принятые решения. Повторная верификация должна учитывать как измененное, так и затронутое программное обеспечение.

Если используется ранее разработанное программное обеспечение, например готовое коммерческое ПО (commercial off-the-shelf, COTS) или повторно используемое ПО, может потребоваться определенная повторная верификация, например если такое ПО устанавливается в новую среду или используется по-новому. План верификации ПО должен это объяснить. Если повторная верификация не требуется, План верификации ПО должен обосновать почему. Дополнительные подробности о верификации приведены в главе 9.

5.3.4 План управления конфигурацией ПО (Software Configuration Management Plan, SCMP)

План управления конфигурацией ПО объясняет, как управлять конфигурацией данных жизненного цикла на протяжении разработки и верификации программного обеспечения. Управление конфигурацией ПО (software configuration management, SCM) начинается на

этапе планирования и продолжается на протяжении всего жизненного цикла ПО, вплоть до развертывания, сопровождения и вывода ПО из эксплуатации.

Раздел 11.4 DO-178C дает сводку того, что должно входить в План управления конфигурацией ПО. Этот план объясняет процедуры, инструменты и методы управления конфигурацией ПО как для инженерного управления конфигурацией, используемого инженерной командой до формальной базовой версии или формальных выпусков, так и для формального управления конфигурацией, а также критерии перехода для этого процесса.

Раздел 7.2 DO-178C объясняет мероприятия управления конфигурацией, которые должны быть подробно описаны в Плане управления конфигурацией ПО. Ниже приведено краткое изложение того, что обычно включается в этот план для каждого такого мероприятия:

1. *Идентификация конфигурации*: План управления конфигурацией ПО объясняет, как каждая единица конфигурации, включая отдельные файлы исходного кода и тестовые файлы, получает уникальную идентификацию. Уникальная идентификация обычно включает нумерацию документов или данных, а также редакции либо версии.
2. *Базовые версии и трассируемость*: План управления конфигурацией ПО объясняет подход к установлению и идентификации базовых версий. Если на протяжении проекта будут устанавливаться инженерные базовые версии, это должно быть объяснено. Аналогично должно быть объяснено установление формальных базовых версий, включая сертификационные и производственные базовые версии. Трассируемость между базовыми версиями также подробно описывается в этом плане.
3. *Регистрация сообщений о проблемах*: процесс регистрации сообщений о проблемах является частью управления конфигурацией и должен быть объяснен в Плане управления конфигурацией ПО, включая то, когда этот процесс начинается, какое содержимое обязательно для сообщения о проблеме (problem report, PR) и как происходят верификация и закрытие такого сообщения. Процесс регистрации сообщений о проблемах критичен для эффективного управления изменениями и должен быть хорошо определен в планах. Инженеры также должны быть обучены этому процессу. Довольно часто план включает форму сообщения о проблеме с кратким пояснением того, как заполнять каждое поле. Большинство сообщений о проблемах содержат поле классификации, чтобы категоризировать серьезность проблемы, и поле состояния, чтобы показывать ее состояние, например “открыто”, “в работе”, “проверено”, “закрыто” или “отложено”. Регистрация сообщений о проблемах рассматривается в главах 9 и 10.

Если для сбора замечаний или действий используется какой-либо другой процесс помимо регистрации сообщений о проблемах, он тоже должен быть объяснен в Плане управления конфигурацией ПО. Это может происходить, когда в компании поверх системы регистрации

сообщений о проблемах существуют дополнительные процессы отслеживания запросов на изменение, отклонений и/или действий.

4. *Управление изменениями*: План управления конфигурацией ПО объясняет, как изменения данных жизненного цикла контролируются таким образом, чтобы причины изменений были установлены и одобрены еще до внесения изменений в элемент данных. Это тесно связано с процессом регистрации сообщений о проблемах.
5. *Рассмотрение изменений*: цель процесса рассмотрения изменений состоит в том, чтобы убедиться, что изменения спланированы, одобрены, задокументированы, правильно реализованы и закрыты. Обычно за этим следит совет по рассмотрению изменений, который одобряет реализацию изменения и обеспечивает, чтобы изменение прошло верификацию до его закрытия. Процесс рассмотрения изменений тесно связан с процессом регистрации сообщений о проблемах.
6. *Учет состояния конфигурации*: на протяжении проекта необходимо знать текущее состояние работ. Учет состояния конфигурации и дает такую возможность. Большинство инструментов управления конфигурацией позволяют формировать отчеты о состоянии. План управления конфигурацией ПО объясняет, что должно входить в эти отчеты, когда они будут формироваться, как они будут использоваться и как инструменты, если они применяются, поддерживают этот процесс. Состояние и классификация сообщений о проблемах на протяжении разработки особенно важны для сертификации. Раздел 7.2.6 DO-178C содержит информацию о том, что следует учитывать в отчетах по учету состояния.
7. *Архивирование, выпуск и извлечение*: у многих компаний существуют подробные процедуры выпуска, архивирования и извлечения. В этом случае на корпоративные процедуры можно сослаться в Плане управления конфигурацией ПО, но необходимо убедиться, что такие процедуры в достаточной мере охватывают руководство DO-178C из разделов 7.2.7 и 11.4.b.7 [1]. Ниже приведены конкретные пункты, которые следует объяснить или указать через ссылку в Плане управления конфигурацией ПО:

- a. *Архивирование*: План управления конфигурацией ПО объясняет, как обеспечивается архивирование.

Обычно это включает архивирование вне основной площадки, а также описание типа носителя, условий хранения и частоты обновления. Следует учитывать долговременную надежность носителя и возможность его считывания.

- b. *Выпуск*: в Плане управления конфигурацией ПО объясняется формальный процесс выпуска данных, которые официально выпускаются. При этом в проекте будут су-

ществовать и данные, которые официально не выпускаются. План управления конфигурацией ПО должен указывать, как такие данные будут храниться.

с. *Извлечение*: кроме того, процессы извлечения должны быть либо объяснены, либо даны по ссылке. Процесс извлечения должен учитывать долговременное извлечение и совместимость носителей, например может оказаться, что для возможности чтения данных в будущем потребуется архивировать и определенное оборудование.

8. *Контроль загрузки программного обеспечения*: План управления конфигурацией ПО объясняет, как ПО точно загружается в целевую систему. Если используется проверка целостности, она должна быть подробно описана. Если такой проверки нет, должен быть определен подход, обеспечивающий точную, полную и неискаженную загрузку.
9. *Контроль среды жизненного цикла программного обеспечения*: среда жизненного цикла ПО, определенная в Плане разработки ПО, Плане верификации ПО и/или указателе конфигурации среды ЖЦ ПО, должна находиться под контролем. Контролируемая среда гарантирует, что все участники команды используют утвержденную среду, и обеспечивает воспроизводимость процесса. План управления конфигурацией ПО описывает, как именно эта среда контролируется. Обычно это включает выпущенный указатель конфигурации среды ЖЦ ПО и оценку, подтверждающую, что перечисленные в нем инструменты полны, точны и действительно используются. Нередко перед формальной сборкой ПО и формальным выполнением тестов требуется аудит конфигурации, или проверка соответствия конфигурации. Гарантия качества ПО может выполнять этот аудит либо присутствовать при его проведении.
10. *Контроль данных жизненного цикла программного обеспечения*: План управления конфигурацией ПО перечисляет все данные жизненного цикла ПО, которые должны быть сформированы, вместе с категоризацией КК1/КК2. Сюда также должно входить объяснение того, как конкретный проект реализует КК1 и КК2 для своих данных. Критерии КК1/КК2 в DO-178C определяют минимально необходимое управление конфигурацией, однако многие проекты идут дальше минимума, объединяют элементы данных или каким-то иным образом отходят от рекомендаций DO-178C. Если информация о КК1/КК2 для каждой единицы конфигурации уже приведена в Плане сертификации ПО, можно сослаться на него, а не повторять ее в Плане управления конфигурацией ПО. Часто План управления конфигурацией ПО перечисляет минимальные назначения КК1/КК2 по DO-178C, а План сертификации ПО дает уже проектно-специфическое назначение. Это позволяет использовать общий общеорганизационный План управления конфигурацией ПО. КК1/КК2 рассматриваются в главе 10.

Если используются поставщики, включая субподрядчиков или офшорные ресурсы, План управления конфигурацией ПО также должен объяснять процесс управления конфигурацией у поставщика. Нередко у поставщика есть собственный План управления конфигурацией ПО, на который и ссылаются. Либо поставщик может следовать Плану управления конфигурацией ПО заказчика. Если используется несколько таких планов, их следует рассматривать на согласованность и совместимость.

План надзора за процессами регистрации сообщений о проблемах у поставщиков также должен быть включен в План управления конфигурацией ПО либо в другой план, на который он ссылается. В разделе 143.а приказа Федерального управления гражданской авиации США (Federal Aviation Administration, FAA) 8110.49 сказано: “Для того чтобы обеспечить последовательную регистрацию и решение проблем программного обеспечения, а также выполнение мероприятий по обеспечению разработки ПО до сертификации, заявителю следует описать в своем Плане управления конфигурацией ПО или в других соответствующих планирующих документах, каким образом он будет осуществлять надзор за процессом регистрации сообщений о проблемах ПО у своего поставщика и поставщиков нижнего уровня” [2]. Далее в этом приказе приводятся ожидания Федерального управления гражданской авиации США к Плану управления конфигурацией ПО, включая объяснение того, как проблемы поставщика будут зарегистрированы, оценены, как будут выработаны и реализованы корректирующие меры, как будет выполнена их повторная верификация (посредством регрессионного тестирования и анализа), а также как они будут закрываться и оставаться под контролем [2]. Европейское агентство по авиационной безопасности (European Aviation Safety Agency, EASA) формулирует аналогичные ожидания в своем документе Certification Memorandum CM-SWCEH-002 [3].

Наконец, План управления конфигурацией ПО определяет и специфические для этого процесса данные, которые будут формироваться, включая сообщения о проблемах, Указатель конфигурации ПО, Указатель конфигурации среды ЖЦ ПО и записи управления конфигурацией. Все эти элементы данных рассматриваются в главе 10.

После чтения десятков Планов управления конфигурацией ПО я выделила несколько часто встречающихся недостатков, и они перечислены здесь просто для вашей бдительности:

- процесс регистрации сообщений о проблемах часто описан недостаточно подробно, чтобы команда разработки и команда верификации могли корректно его реализовать; кроме того, может быть неясно, когда именно этот процесс должен стартовать;
- процесс контроля среды редко бывает определен должным образом; как следствие, во многих проектах отсутствует адекватный контроль среды;
- не описывается подход к архивированию, в том числе как будет архивироваться сама

среда, то есть инструменты, использовавшиеся для разработки, верификации, конфигурирования и управления программным обеспечением;

- инженерный процесс управления конфигурацией, которым инженеры пользуются ежедневно, редко бывает расписан достаточно подробно;
- контроль поставщиков часто объяснен в Плане управления конфигурацией ПО недостаточно, а различные процессы управления конфигурацией между компаниями редко рассматриваются на согласованность;
- план установления инженерных базовых версий обычно не раскрывается достаточно подробно; инструменты управления конфигурацией ПО часто не идентифицируются и не ставятся под контроль.

Во многих компаниях существует общеорганизационный План управления конфигурацией ПО. В этом случае проектно-специфические детали все равно должны быть где-то отражены. Это может быть отдельный проектно-специфический План управления конфигурацией ПО, дополняющий общий корпоративный план, либо План сертификации ПО, либо какой-то другой логичный документ. Как бы это ни было оформлено, это должно быть ясно из Плана сертификации ПО, а также из Плана разработки ПО и Плана верификации ПО, чтобы участники команды понимали и соблюдали утвержденные процессы.

5.3.5 План гарантии качества ПО (Software Quality Assurance Plan, SQAP)

План гарантии качества ПО описывает мероприятия команды качества программного обеспечения, направленные на то, чтобы ПО соответствовало утвержденным планам и стандартам, а также целям DO-178C. Этот план включает место команды гарантии качества ПО в общей структуре компании и подчеркивает ее независимость.

План гарантии качества ПО также объясняет роль инженера по качеству программного обеспечения[*], которая обычно включает следующее:

- рассмотрение планов и стандартов;
- участие в процессе коллегиального рассмотрения, чтобы удостовериться, что этот процесс выполняется надлежащим образом;
- обеспечение соблюдения критериев перехода, определенных в планах;
- аудит среды, чтобы убедиться, что разработчики и верификаторы используют инструменты, указанные в указателе конфигурации среды ЖЦ ПО, с надлежащими настройками, включая компилятор, компоновщик, тестовые инструменты и оборудование;
- оценку соответствия планам;
- присутствие при сборках и тестах программного обеспечения;
- подписание или утверждение ключевых документов;
- закрытие сообщений о проблемах;

- участие в совете по управлению изменениями;
- выполнение рассмотрения соответствия ПО.

Гарантия качества ПО дополнительно рассматривается в главе 11. Выполняя свои обязанности, инженеры по качеству программного обеспечения формируют записи гарантии качества ПО, в которых объясняется, что именно они делали и какие несоответствия были обнаружены. Нередко одна и та же форма используется для нескольких мероприятий гарантии качества ПО, а пустая форма включается в План гарантии качества ПО. План гарантии качества ПО должен объяснять подход к ведению таких записей, включая то, какие записи будут сохраняться, где они будут храниться и как будет управляться их конфигурация.

Во многих случаях План гарантии качества ПО задает целевые проценты участия качества. Это может включать процент участия в коллегиальных рассмотрениях и процент присутствия на тестах. Если это так, должно быть ясно, как будут собираться соответствующие метрики.

План гарантии качества ПО должен объяснять критерии перехода для процесса гарантии качества ПО, то есть когда начинаются соответствующие мероприятия, а также любые ключевые временные детали того, когда они будут выполняться.

Во многих компаниях помимо гарантии качества ПО внедрена гарантия качества продукта (product quality assurance, PQA). Инженер по качеству продукта (product quality engineer, PQE) наблюдает за повседневной работой проекта и фокусируется как на техническом качестве, так и на соблюдении процессов. Если гарантия качества продукта используется для удовлетворения каких-либо целей таблицы A-9 DO-178C, План гарантии качества ПО должен описывать ее роль и то, как она координируется с ролью гарантии качества ПО.

Если поставщики помогают в разработке или верификации программного обеспечения, План гарантии качества ПО должен объяснять, как гарантия качества ПО будет осуществлять надзор за ними. Если у поставщика есть собственная команда гарантии качества ПО и собственные планы в этой области, их следует оценить на согласованность с Планом гарантии качества ПО более высокого уровня.

Следует отметить, что если План гарантии качества ПО является общеорганизационным планом, необходимо убедиться, что он согласован с планами конкретного проекта. Нередко у проекта есть специфические потребности, которые не упомянуты в общеорганизационном плане. Если это так, для покрытия проектно-специфических потребностей в гарантии качества ПО могут использоваться отдельный План гарантии качества ПО, План сертификации ПО или дополнение к Плану гарантии качества ПО.

5.4 Три стандарта разработки

В дополнение к пяти планам DO-178C указывает на необходимость трех стандартов: стандарта требований, стандарта проектирования и стандарта кодирования. Многие компании, впервые сталкивающиеся с DO-178C, путают их с общеотраслевыми стандартами (например, со стандартами Института инженеров по электротехнике и электронике (Institute of Electrical and Electronic Engineers)). Общеотраслевые стандарты могут служить входом для проектно-специфических стандартов, однако у каждого проекта есть собственные потребности, которые в них не рассматриваются.

Если в нескольких проектах используются одни и те же методы и языки, могут быть созданы и применены общеорганизационные стандарты. Однако применимость таких стандартов должна оцениваться для каждого проекта. Стандарты задают правила и ограничения, помогающие разработчикам выполнять свою работу правильно и избегать действий, которые могут иметь отрицательные последствия для безопасности или функциональности.

Многие стандарты оказываются неэффективными по следующим причинам:

- стандарты создаются лишь для галочки ради соответствия DO-178C и имеют мало реальной ценности для проекта;
- стандарты вырезаются и вставляются из отраслевых документов, но не удовлетворяют потребностям конкретного проекта;
- стандарты не читаются до этапов верификации (разработчики либо не знают о них, либо игнорируют их).

Цель этого раздела – дать несколько практических рекомендаций по разработке эффективных стандартов.

Хотя это и не требуется, большинство стандартов включает как обязательный, так и рекомендательный материал. Обязательные пункты обычно называют *rules* (правилами), а рекомендательные – *guidelines* (рекомендациями). Во многих случаях применимость правил и рекомендаций различается в зависимости от уровня программного обеспечения (например, один и тот же пункт может быть рекомендательным для уровня C, но обязательным для уровней A и B). Следует отметить, что DO-178C не требует стандартов для уровня D, хотя и не запрещает их.

Прежде чем переходить непосредственно к стандартам, хочу подчеркнуть их важность. Они служат инструкциями для разработчиков по применению безопасных и эффективных практик. Если стандарты написаны ясно и содержат обоснование и хорошие примеры, они могут стать отличным ресурсом для разработчиков. После разработки стандартов важно провести обучение для всех разработчиков. Это обучение должно охватывать содержание

стандартов, объяснять, как ими пользоваться, и подчеркивать, что чтение и соблюдение стандартов обязательно.

5.4.1 Стандарты требований к программному обеспечению

Стандарты требований к программному обеспечению определяют методы, инструменты, правила и ограничения для разработки требований к ПО [1]. Обычно они применяются к требованиям высокого уровня; однако в некоторых проектах стандарты требований также применяются к требованиям низкого уровня. В общем случае стандарты требований являются руководством для команды, пишущей требования. Например, они объясняют, как писать эффективные и реализуемые требования, как использовать инструмент управления требованиями, как выполнять трассируемость, как обрабатывать производные требования и как создавать требования, удовлетворяющие критериям DO-178C. Стандарты требований могут также служить инструментом обучения инженеров и одновременно задавать критерии успешности для рассмотрения требований.

Ниже приведен перечень элементов, которые обычно включаются в стандарты требований:

- критерии из таблицы A-3 DO-178C (чтобы проактивно учитывать ожидания DO-178C);
- определения и примеры требований высокого уровня, требований низкого уровня и производных требований (в справочных целях);
- качественные атрибуты требований (верифицируемость, однозначность, согласованность и т.д.). В главах 2 (раздел 2.2.3) и 6 (раздел 6.7.6.9) объясняются характеристики хороших требований;
- подход к трассируемости и соответствующие инструкции;
- критерии использования инструмента управления требованиями (включая объяснение каждого атрибута и указания относительно обязательной и необязательной информации);
- критерии идентификации требований (например, схема нумерации или запрет на повторное использование номера);
- если требования представляются в виде таблиц, объяснение того, как правильно их использовать и идентифицировать (например, нумеровать каждую строку или столбец);
- если для представления или дополнения требований используется графика, описание того, как использовать каждый тип графики и любые символы. Кроме того, может потребоваться определить способ идентификации и трассировки каждого блока или символа. О модельно-ориентированной разработке см. главу 14;
- критерии различения требований и пояснительного материала;
- критерии документирования производных требований (включая обоснование производных требований, чтобы помочь специалистам по безопасности в их оценке безопас-

ности);

- ограничения или лимиты на используемые инструменты;
- критерии разработки робастных требований;
- критерии обработки допусков внутри требований;
- критерии использования документов управления интерфейсами и документирования требований, которые на них ссылаются;
- примеры применения правил и рекомендаций.

Глава 6 посвящена программным требованиям. Некоторые концепции, обсуждаемые в главе 6, могут быть уместны для включения в стандарты требований.

5.4.2 Стандарты проектирования программного обеспечения

Стандарты проектирования программного обеспечения определяют методы, инструменты, правила и ограничения для разработки проекта ПО [1]. В DO-178C проект ПО включает требования низкого уровня и архитектуру ПО.

Стандарты проектирования служат руководством для команды, разрабатывающей проект ПО. Эти стандарты объясняют, как создавать эффективный и реализуемый проект ПО, как использовать инструменты проектирования, как выполнять трассируемость, как обрабатывать производные требования низкого уровня и как создавать проектные данные, удовлетворяющие критериям DO-178C. Стандарты проектирования могут также служить инструментом обучения инженеров и одновременно задавать критерии для рассмотрения проектных данных.

Поскольку проектные данные могут представляться разными способами, разработать универсальный стандарт проектирования трудно. У многих компаний есть общие стандарты проектирования, но они часто оказываются неэффективными для нужд конкретного проекта. Каждый проект должен определить желаемую методологию и предоставить проектировщикам инструкции по ее правильному применению. Может оказаться возможным использовать общеорганизационные стандарты как исходную точку, но зачастую требуется определенная адаптация. Если адаптация незначительна, ее может быть целесообразно обсудить в Плане разработки ПО, а не обновлять сам стандарт.

Ниже приведен перечень элементов, которые обычно включаются в стандарты проектирования:

- критерии из таблицы A-4 DO-178C (чтобы проактивно учитывать цели DO-178C);
- предпочтительная структура проектного документа;
- критерии для требований низкого уровня (требования низкого уровня будут обладать теми же качественными атрибутами, что и требования высокого уровня; однако требования низкого уровня относятся к проектированию и описывают *как*, а не *что*);

- подход к трассируемости между требованиями высокого уровня и требованиями низкого уровня;
- критерии документирования производных требований низкого уровня и их обоснования;
- рекомендации по эффективной архитектуре, которые могут включать блочные диаграммы, структурные схемы, диаграммы переходов состояний, диаграммы потоков управления и данных, блок-схемы, деревья вызовов, диаграммы “сущность-связь” и т.д.;
- соглашения об именовании модулей, которые должны быть согласованы с тем, что затем будет реализовано в коде;
- проектные ограничения (например, ограничение уровня вложенности условий либо запрет рекурсивных функций, безусловных переходов, обработчиков прерываний, которые могут быть вызваны повторно до завершения предыдущего вызова, и кода, который изменяет сам себя во время выполнения);
- рекомендации по робастному проектированию;
- указания о том, как документировать отключенный код в проектных данных (информация об отключенном коде приведена в главе 17).

Глава 7 содержит рекомендации по хорошему проектированию программного обеспечения. Некоторые из концепций, обсуждаемых в главе 7, могут быть уместны для включения в стандарты проектирования.

5.4.3 Стандарты кодирования программного обеспечения

Как и стандарты требований и проектирования, стандарты кодирования используются для предоставления рекомендаций программистам. Стандарты кодирования объясняют, как правильно использовать конкретный язык, ограничивают некоторые конструкции языка, которые может быть нежелательно применять в области программного обеспечения, критичного по безопасности, определяют соглашения об именовании, объясняют использование глобальных данных и помогают создавать читаемый и сопровождаемый код.

Стандарты кодирования довольно обычны в разработке программного обеспечения, и при их создании существуют полезные общепромышленные ресурсы. Например, стандарт C от Motor Industry Software Reliability Association (MISRA-C) – отличный стандарт, который стоит рассмотреть как вход для разработки стандартов кодирования на C.

Стандарты кодирования зависят от языка. Если в проекте используется несколько языков, каждый язык должен быть рассмотрен в стандартах. Даже для языка ассемблера должны существовать рекомендации по использованию.

Ниже приведен перечень элементов, которые обычно включаются в стандарты кодирования:

- критерии из таблицы A-5 DO-178C (для проактивного учета целей);
- подход к документированию трассируемости между кодом и требованиями низкого уровня;
- соглашения об именовании модулей и функций либо процедур;
- рекомендации по использованию локальных и глобальных данных;
- рекомендации по читаемости и сопровождаемости кода (например, использование комментариев и пробельных символов, применение абстракции, ограничение размера файла и ограничение глубины вложенных условий);
- указания по структуре модуля (включая формат заголовка и разделы модуля);
- рекомендации по проектированию функций (например, формат заголовка, структура функции, уникальная идентификация/имя функции, требование, чтобы функции не были рекурсивными и не вызывались повторно до завершения предыдущего вызова, правила входа и выхода);
- ограничения для условно компилируемого кода;
- рекомендации по использованию макросов;
- другие ограничения (например, ограничение или запрет на использование указателей, запрет кода, который может вызываться повторно до завершения предыдущего вызова, и рекурсивного кода).

Полезно включать в стандарты кодирования обоснование и примеры для каждой из указанных рекомендаций. Когда программист понимает, почему что-то желательно или запрещено, ему или ей легче применять эти указания.

Глава 8 содержит рекомендации по написанию кода, критичного по безопасности. Некоторые из концепций, обсуждаемых в главе 8, могут быть уместны для включения в стандарты кодирования.

5.5 Планирование квалификации инструментов

Если инструмент должен быть квалифицирован, это требует определенного планирования. Плановая информация может быть включена в План сертификации ПО; однако для инструментов с более высокими уровнями квалификации или для инструментов, которые будут повторно использоваться в других проектах, могут потребоваться дополнительные планы по инструментам. Квалификация инструментов рассматривается в главе 13. Пока же достаточно понимать, что квалификация инструментов требует особого планирования.

5.6 Другие планы

В дополнение к пяти планам, указанным в DO-178C, у проекта могут быть и другие планы, помогающие в управлении проектом. Они могут быть менее формальными, но все равно важны для общего управления проектом. В этом разделе рассматриваются три таких плана.

5.6.1 План управления проектом

Этот план определяет состав команды и обязанности (включая имена и конкретные назначения), детальный график, состояние работ, собираемые метрики и т.д. Он содержит больше деталей, чем План разработки ПО и План верификации ПО, относительно организационных деталей проекта и предоставляет средство убедиться, что все управляется надлежащим образом.

5.6.2 План управления требованиями

Этот план иногда разрабатывается в дополнение к Плану разработки ПО и стандартам требований к программному обеспечению, чтобы обеспечить надлежащее документирование, распределение и трассировку требований. Он также может помогать обеспечивать согласованное документирование требований между командами по системам, аппаратному обеспечению, безопасности и ПО.

5.6.3 План тестов

Этот план объясняет детали стратегии тестирования, включая необходимое оборудование, процедуры, инструменты, формат тестовых примеров, стратегию трассировки и т.д. Он может быть частью документа “Примеры и процедуры верификации” (Software Verification Cases and Procedures, SVCP) либо отдельным самостоятельным планом. План тестов часто включает контрольные перечни и критерии рассмотрения готовности к тестам, которые команда будет использовать, чтобы убедиться, что она готова начать формальное выполнение тестов. Планирование тестов рассматривается в главе 9.

Литература

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
2. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Change 1, September 2011).
3. European Aviation Safety Agency, *Software Aspects of Certification*, Certification Memorandum CM-SWCEH-002 (Issue 1, August 2011).

*В главе 4 приведен обзор технологических дополнений. Главы 14-16 содержат более подробную информацию по каждому из технологических дополнений.

†План сертификации ПО (Plan for Software Aspects of Certification, PSAC) – это верхнеуровневый план по ПО, который представляется сертифицирующему органу. План сертификации ПО рассматривается далее в этой главе.

*Уполномоченные представители – это представители сертифицирующего органа или делегированной организации, которые уполномочены рассматривать данные и либо утверждать их, либо рекомендовать к утверждению. Проекты обычно координируются с такими представителями до взаимодействия с сертифицирующим органом.

*Итоговый отчёт разработки ПО (Software Accomplishment Summary, SAS) – это отчёт о соответствии для ПО, соответствующего DO-178C, который представляется сертифицирующему органу. Он составляется в конце программного проекта и указывает отклонения от планов. Итоговый отчёт разработки ПО рассматривается в главе 12.

- *Big bang* – это гипотетическая модель, в которой в конце проекта все магическим образом появляется сразу. Модели *tornado* и *smoke-and-mirrors* – это авторские неологизмы Лианны. Я использую *tornado model* для описания проекта, идущего к катастрофе, а *smoke-and-mirrors model* – для описания проекта, у которого на самом деле нет никакого плана.

*В большинстве программных проектов назначается один инженер по качеству программного обеспечения. Однако на крупных или распределенных проектах инженеров по качеству может быть несколько.

Глава 6. Требования к программному обеспечению

Сокращения

Сокращение	Расшифровка
CASE	Computer-aided software engineering (автоматизированная разработка программного обеспечения)
CAST	Certification Authorities Software Team (группа органов сертификации по ПО)
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
FAQ	Frequently asked question (часто задаваемый вопрос)
TBY	HLR, high-level requirements (требования высокого уровня к ПО)
IEEE	Institute of Electrical and Electronic Engineers (Институт инженеров по электротехнике и электронике)
THY	LLR, low-level requirements (требования низкого уровня к ПО)
PR	Сообщение о проблеме (problem report)
SWRD	Документ требований высокого уровня к ПО (Software Requirements Document)
TBD	To be determined (подлежит определению)

6.1 Введение

Требования к программному обеспечению являются фундаментом соответствия DO-178C и разработки ПО, критичного по безопасности. Успех или провал проекта зависят от качества требований. Как пишет Нэнси Левесон:

Подавляющее большинство аварий, в которых было задействовано программное обеспечение, можно проследить до дефектов требований и, более конкретно, до неполноты заданного и реализованного поведения ПО, то есть до неполных или неверных предположений о работе управляемой системы или требуемой работе компьютера и необработанных состояний управляемой системы и условий окружающей среды. Хотя ошибки кодирования часто привлекают наибольшее внимание, они сильнее влияют на надежность и другие качества, чем на безопасность [1].

Каковы требования, таков и проект. Самые хаотичные проекты, которые мне доводилось пережить или наблюдать, начинались с плохих требований и дальше только ухудшались. И наоборот, лучшие проекты, которые я видела, это те, где были приложены усилия к тому, чтобы сделать требования правильными. Несколько основ эффективных требований уже

были рассмотрены в главе 2 при обсуждении системных требований. Поэтому я рекомендую вам прочитать или перечитать раздел 2.2, если вы давно этого не делали. Эта глава развивает идеи раздела 2.2, делая акцент на требованиях к программному обеспечению, а не на системных требованиях.

Многие положения, обсуждаемые в этой главе, применимы также и к системным требованиям и могут дополнить материал, представленный в главе 2. Граница между системными требованиями и требованиями к программному обеспечению часто бывает очень размытой. В общем случае требования к ПО уточняют валидированные системные требования и используются разработчиками ПО для проектирования и реализации ПО. Кроме того, требования к ПО определяют то, что делает ПО, а не то, что делает система. При написании требований к ПО могут быть выявлены ошибки, недостатки и упущения в системных требованиях; такие проблемы следует документировать в сообщениях о проблемах (problem report, PR) и устранять силами системной команды.

В этой главе рассматриваются важность хороших требований и то, как писать, верифицировать и управлять требованиями. Кроме того, глава завершается обсуждением прототипирования и трассируемости, двух тем, тесно связанных с разработкой требований.

6.2 Определение требования

Институт инженеров по электротехнике и электронике (IEEE) определяет *требование* следующим образом [2]:

1. Условие или возможность, необходимые пользователю для решения проблемы или достижения цели.
2. Условие или возможность, которые должны быть выполнены системой или компонентом системы либо которыми они должны обладать, чтобы удовлетворять контракту, стандарту, спецификации или другому формально установленному документу.
3. Документированное представление условия или возможности, как в пунктах 1 или 2.

Глоссарий DO-178C определяет *software requirement* (требование к программному обеспечению), *high-level requirements* (требования высокого уровня), *low-level requirements* (требования низкого уровня) и *derived requirements* (производные требования) следующим образом [3]:

- *Software requirement* – «Описание того, что должно порождать ПО при заданных входных данных и заданных ограничениях. Требования к ПО включают требования высокого и низкого уровней».
- *High-level requirements* – «Требования к ПО, разработанные на основе анализа требований к системе, требований к безопасности и архитектуры системы».

- *Low-level requirements* – «Требования к ПО, разработанные на основе требований высокого уровня, производных требований и проектных ограничений, на основе которых можно непосредственно реализовать исходный код без привлечения дополнительной информации».
- *Derived requirements* – «Требования, создаваемые процессами разработки ПО, которые (а) не трассируются непосредственно на требования более высокого уровня и/или (б) описывают особенности поведения, которые не определяются требованиями к системе или требованиями к ПО более высокого уровня».

В отличие от определения IEEE, DO-178C определяет два уровня требований к программному обеспечению: требования высокого уровня к ПО (*high-level requirements*) и требования низкого уровня к ПО (*low-level requirements*). Эта глава сосредоточена на требованиях высокого уровня к ПО, которые далее в этой главе для простоты будут именоваться просто *требованиями*. DO-178C относит требования низкого уровня к проектированию, поэтому они будут обсуждаться в следующей главе.

В общем случае требования предназначены для того, чтобы «описать, что у нас будет, когда проект будет завершен» [4]. Требования к программному обеспечению обычно охватывают следующее: функциональность, внешние интерфейсы, производительность, атрибуты качества (например, переносимость или сопровождаемость), проектные ограничения, безопасность и защиту.

Хорошие требования не затрагивают детали проектирования или реализации, детали управления проектом (такие как стоимость, график, методология разработки) или детали испытаний.

6.3 Важность хороших требований к программному обеспечению

Рассмотрим пять причин, по которым требования так важны для разработки программного обеспечения, критичного по безопасности.

6.3.1 Причина 1: Требования являются фундаментом разработки программного обеспечения

Я не архитектор, но здравый смысл подсказывает мне, что при строительстве дома фундамент чрезвычайно важен. Если фундамент сделан из слабого или дефектного материала, если в нем отсутствуют части или если он неровный, у построенного на нем дома будут долговременные проблемы. То же верно и для разработки программного обеспечения. Если требования (фундамент) слабы, это имеет долгосрочные последствия и приводит к неопишимым проблемам и трудностям для всех участников, включая заказчика.

Размышляя о проектах, которые мне удалось пережить за эти годы, я вижу некоторые об-

щие характеристики, приводившие к плохим требованиям. Во-первых, требования к программному обеспечению разрабатывались неопытными командами. Во-вторых, команды подталкивали к тому, чтобы как можно быстрее поставить заказчику хоть что-нибудь, еще до того, как сами команды были к этому готовы. В-третьих, системные требования не были валидированы до передачи их в разработку ПО. Эти особенности приводили к следующим типичным результатам:

- Заказчики оставались недовольны.
- В продуктах возникало чрезвычайно большое число сообщений о проблемах.
- Программное обеспечение требовало по меньшей мере одного полного перепроектирования (а в паре случаев потребовались еще две дополнительные итерации).
- Проекты значительно выходили за пределы сроков и бюджета.
- Несколько руководителей переводили на другие должности (прямо на другие обреченные проекты), а их карьера оказывалась под ударом.

Результаты плохих требований приводят к тому, что я называю *эффектом снежного кома*. Проблемы и сложность накапливаются до тех пор, пока проект не превращается в огромный, неуправляемый *снежный ком*. На рисунке 6.1 показана эта проблема.



Рисунок 6.1 Эффекты плохих требований и недостаточного процесса рассмотрения.

Когда артефакты разработки не рассматриваются и не доводятся до зрелости перед переходом к следующей фазе разработки, выявить и устранить ошибку или ошибки позднее становится труднее и дороже. В некоторых случаях может даже стать невозможным выявить ошибку, поскольку первопричина оказывается настолько глубоко погребена в данных (в этом снежном коме). Все этапы разработки итеративны и подвержены изменениям по мере продвижения проекта; однако построение последующих работ разработки на неполных и ошибочных входных данных является одной из самых частых ошибок и неэффективностей в программной инженерии.

Требования никогда не будут идеальными с первого раза, но цель состоит в том, чтобы сделать хотя бы часть требований максимально полной и точной, чтобы можно было переходить к проектированию и реализации при приемлемом уровне риска. Со временем требования обновляются, чтобы добавить или изменить функциональность и внести изменения на основе зрелости проектирования и реализации. Такой итеративный подход является

наиболее распространенным способом одновременно получить качественные требования и удовлетворить потребности заказчика. В книге *Software Requirements* Карл Вигерс пишет: Итеративность является ключом к успеху разработки требований. Планируйте несколько циклов исследования требований, уточнения требований высокого уровня до деталей и подтверждения их корректности с пользователями. Это требует времени и может вызывать раздражение, но это неотъемлемая часть работы с нечеткой неопределенностью, сопровождающей определение нового программного продукта [4].

6.3.2 Причина 2: Хорошие требования экономят время и деньги

Эффективная инженерия требований, вероятно, является вложением с наибольшей отдачей из всех, какие проект вообще может сделать. Многочисленные исследования показывают, что самыми дорогими являются ошибки, зародившиеся на фазе требований, и что крупнейшей причиной переделок программного обеспечения являются плохие требования. Одно из исследований даже показало, что «ошибки требований составляют от 70 до 85 процентов стоимости переделок» [5].

Исследование Standish Group привело следующие причины неудач программных проектов (несколько из них связаны с требованиями) [6]:

- Неполные требования – 13,1%
- Недостаточная вовлеченность пользователей – 12,4%
- Недостаток ресурсов или времени по графику – 10,6%
- Нереалистичные ожидания – 9,9%
- Недостаточная поддержка со стороны руководства – 9,3%
- Изменяющиеся требования – 8,7%
- Плохое планирование – 8,1%
- Программное обеспечение больше не требуется – 7,4%

Хотя это исследование уже не новое, его результаты соответствуют тому, что я вижу в авиационных проектах из года в год. Хорошие требования необходимы для получения успешного результата. Меня не перестает удивлять, как много проектов не находят времени, чтобы сделать работу правильно с первого раза, но в итоге находят время и деньги, чтобы сделать ее два или три раза.

6.3.3 Причина 3: Хорошие требования необходимы для безопасности

В исследовательском отчете Федерального управления гражданской авиации США (FAA) под названием *Requirements Engineering Management Findings Report* говорится: “Исследо-

ватели, сосредоточенные на системах, критичных по безопасности, установили, что ошибки в требованиях с большей вероятностью влияют на безопасность встроенной системы, чем ошибки, внесенные на этапе проектирования или реализации” [7]. Без хороших требований невозможно выполнить нормативные требования. Федеральное управление гражданской авиации США и другие регулирующие органы по всему миру имеют юридически обязательные нормы, требующие, чтобы для каждой системы самолета было показано, что она выполняет свою предназначенную функцию при любых предсказуемых условиях эксплуатации. Это означает, что предназначенные функции должны быть определены и доказаны. Требования являются формальным способом передачи соображений безопасности и предусмотренной функциональности.

6.3.4 Причина 4: Хорошие требования необходимы для удовлетворения потребностей заказчика

Без точных и полных требований ожидания заказчика не будут удовлетворены. Требования используются для коммуникации между заказчиком и разработчиком. Плохие требования означают плохую коммуникацию, а это обычно ведет к плохому продукту.

Один проект уровня А, который я оценивала много лет назад, имел ужасные требования к программному обеспечению и не имел никаких шансов вовремя завершить работы уровня А, чтобы поддержать график самолета. Из-за недостатков ПО заказчику пришлось переработать часть самолета, чтобы отключать систему при критичных по безопасности режимах эксплуатации, добавить аппаратные средства для компенсации того, что должно было делать ПО, и снизить зависимость от ПО до уровня D. После первичной сертификации был выбран новый поставщик, а прежний поставщик был уволен. Основные причины этого провала были следующими: (1) требования к ПО не соответствовали системным требованиям заказчика и (2) когда системные требования были неясны, команда ПО просто импровизировала вместо того, чтобы спрашивать заказчика, что именно ему нужно. Конечно, у системных требований тоже были свои проблемы, но всей этой катастрофы можно было избежать при лучшей коммуникации и лучшей разработке требований в обеих компаниях.

6.3.5 Причина 5: Хорошие требования важны для тестирования

Требования определяют работы по тестированию. Если требования написаны плохо или неполны, возможно следующее:

- В результате тесты, основанные на требованиях, будут проверять не то и/или неполно проверять то, что нужно.
- На этапе тестирования могут потребоваться значительные усилия, чтобы выявить и установить *реальные* требования.
- Будет трудно доказать наличие предусмотренной функциональности.

- Будет сложно или невозможно доказать отсутствие непредусмотренной функциональности (то есть показать, что программное обеспечение делает то, что должно, и только то, что должно).

Безопасность во многом зависит от способности показать, что предусмотренные функции выполняются и что никакая непредусмотренная функциональность не окажет влияния на безопасность.

6.4 Инженер по требованиям к программному обеспечению

Разработка требований обычно выполняется одним или несколькими инженерами по требованиям (их также называют аналитиками требований).[*] В большинстве успешных проектов есть по меньшей мере два ведущих инженера по требованиям, которые тесно работают вместе на протяжении всего проекта. Они совместно разрабатывают требования и постоянно рассматривают работу друг друга по мере ее продвижения. Они также постоянно выполняют *проверки здравого смысла*, чтобы убедиться, что требования согласованы между собой и реализуемы. Всегда полезно, когда вместе со старшими инженерами работают и младшие инженеры, поскольку эти знания понадобятся в будущих проектах, а организациям нужно выращивать будущих экспертов по требованиям. Организациям следует внимательно подходить к выбору инженеров, которым доверяется разработка требований; не каждый способен разрабатывать и документировать хорошие требования. Навыки, необходимые эффективному инженеру по требованиям, рассматриваются ниже.

Навык 1: Опыт написания требований. Ничто не заменит опыт. Человек, прошедший через несколько проектов, знает, что работает, а что нет. Конечно, лучше, если этот опыт основан на успешных проектах, но и не слишком успешные проекты тоже дают опыт, если человек готов учиться на ошибках.

Навык 2: Умение работать в команде. Поскольку инженеры по требованиям взаимодействуют почти со всеми участниками команды, важно, чтобы они умели работать в коллективе и хорошо ладили с другими. Инженер по требованиям к программному обеспечению будет тесно взаимодействовать с системными инженерами (а возможно, и с заказчиками), проектировщиками и программистами, руководителем проекта, службой качества и тестировщиками. Одиночки и эгоманьяки плохо подходят для такой работы и часто не руководствуются лучшими интересами команды.

Навык 3: Умение слушать и наблюдать. Инженерия требований часто связана с тем, чтобы улавливать тонкие сигналы от системных инженеров или заказчика. Она требует способности замечать и учитывать отсутствующие элементы; важно не только понимать, что уже есть, но и определять, чего нет и что должно быть.

Навык 4: Внимание и к общей картине, и к деталям. Инженер по требованиям должен не только понимать, как программное обеспечение, аппаратные средства и система сочетаются друг с другом, но и уметь представлять себе и документировать детали. Инженер по требованиям должен уметь мыслить как сверху вниз, так и снизу вверх.

Навык 5: Навыки письменной коммуникации. Очевидно, одна из главных ролей инженера по требованиям состоит в документировании требований. Он или она должны уметь писать ясно и организованно. Успешны те инженеры по требованиям, которые способны понятно излагать сложные идеи и проблемы. Кроме того, инженер по требованиям должен уверенно использовать графические средства для передачи идей, которые трудно объяснить текстом. Примеры применяемых графических средств: таблицы, блок-схемы, диаграммы потоков данных, диаграммы потоков управления, варианты использования, диаграммы состояний, диаграммы типа statechart и диаграммы последовательностей и времени.

Навык 6: Преданность делу. Ведущий инженер или ведущие инженеры по требованиям должны быть людьми, настроенными довести проект до конца. Я видела, как несколько проектов страдали из-за того, что ведущий инженер уходил на другую работу и уносил с собой значительную часть жизненно важных знаний у себя в голове. Это еще одна причина, по которой я рекомендую командный подход к разработке требований.

Навык 7: Опыт в предметной области. Желательно, чтобы среди инженеров по требованиям был человек, хорошо знающий предметную область. Человек с опытом в навигационных системах может не знать тонкостей, необходимых для задания требований к тормозной или топливной системе. Хотя это и несколько субъективно, я считаю, что лучше всего, когда по меньшей мере половина команды разработки имеет опыт в соответствующей предметной области.

Навык 8: Творческий подход. Требования, вымученные грубой силой, обычно оказываются не самыми эффективными. Хорошие требования – это смесь искусства и науки. Требуется творческое мышление (способность *мыслить вне рамок*) для разработки оптимальных требований, которые охватывают предусмотренные функции и предотвращают непредусмотренные.

Навык 9: Организованность. Если инженер по требованиям неорганизован, результат его работы может не передавать должным образом то, что предполагалось. Кроме того, поскольку инженеры по требованиям критичны для проекта, их нередко отвлекают менее важными задачами; поэтому они должны уметь организовывать работу, расставлять приоритеты и сохранять фокус.

6.5 Обзор разработки требований к программному обеспечению

В главе 5 рассматривался процесс планирования, предшествующий разработке требований. При планировании необходимо учитывать несколько аспектов, связанных с требованиями, включая методологию и формат требований, использование средств управления требованиями, определение состава команды разработки, процесс рассмотрения требований, стратегию трассировки, определение стандартов требований и т.д. Планирование необходимо для эффективной разработки требований.

Работы по разработке требований можно организовать в виде семи мероприятий:

1. сбор и анализ входных данных;
2. написание требований;
3. рассмотрение требований;
4. установление базовой версии, выпуск и архивирование требований;
5. реализация требований;
6. тестирование требований;
7. изменение требований с использованием процесса управления изменениями.

На рисунке 6.2 показаны эти мероприятия. Мероприятия 1-3 рассматриваются в этой главе; мероприятия 4 и 7 рассматриваются в главе 10; мероприятие 5 обсуждается в главах 7 и 8; а мероприятие 6 рассматривается в главе 9.



Рисунок 6.2 Мероприятия по разработке требований.

Каждый проект подходит к разработке системных требований и требований к программно-

му обеспечению немного по-своему. В таблице 6.1 приведены наиболее распространенные подходы к разработке системных требований и требований к ПО, а также преимущества и недостатки каждого подхода. По мере того как все больше системных команд и сертифицирующих органов будут принимать ARP4754A, состояние системных требований должно улучшаться. Высококачественные требования зависят от эффективной коммуникации и партнерства между заинтересованными сторонами, включая заказчика, системную команду (включая персонал по безопасности), команду аппаратного обеспечения и команду ПО.

Таблица 6.1 Подходы к разработке системных требований и требований к программному обеспечению

Подход	Преимущества	Недостатки
<i>Продукт, определяемый поставщиком:</i> системные требования и требования к программному обеспечению разрабатываются одной и той же компанией.	• Способствует более открытому общению. • Обычно означает, что и системная часть, и программная часть имеют один и тот же или очень похожий процесс рассмотрения и выпуска. • Предметная экспертиза находится у поставщика. • Есть потенциал для широкого повторного использования и повторного применения.	• Заказчик и системные команды могут не задавать вещи с достаточной степенью детализации. • Команды могут быть не размещены совместно, даже если находятся в одной компании. • Часто команда ПО более дисциплинирована в документировании требований, чем системная команда. • Может присутствовать подход по принципу "перекинули через стену". • Системные требования могут быть написаны на неправильном уровне, быть чрезмерно предписывающими и ограничивать варианты проектирования ПО. • Заказчик может медленно обновлять системные требования или сопротивляться их обновлению, когда обнаруживаются недостатки. • У поставщика может не быть возможности протестировать все требования.
<i>Продукт, определяемый заказчиком:</i> системные требования разрабатываются заказчиком и передаются поставщику для программной реализации.	• Часто требования заказчика очень детальны, когда предполагается аутсорсинг. • Позволяет заказчику выбрать поставщика с нужной предметной экспертизой, которой может не быть внутри самой организации.	
<i>Объединенные требования:</i> системные требования и требования к ПО объединяются в один уровень одной и той же компанией.	• Обеспечивает согласованность между системными требованиями и требованиями к ПО. • Есть потенциал для меньшего дублирования работ по тестированию. • Если продукт прост, это может устранить искусственные уровни требований.	• Это затрудняет достижение надлежащего уровня детализации и степени подробности в требованиях. • Это может вынудить делать системные требования слишком подробными или, наоборот, оставить требования к ПО на ненадлежащим образом высоком уровне. • Это может вызывать проблемы при распределении требований между аппаратным и ПО. • За исключением простого продукта, трудно показать сертифицирующему органу, что все цели охвачены.

6.6 Сбор и анализ входных данных для требований к программному обеспечению

DO-178C исходит из того, что системные требования, передаваемые команде программного обеспечения, полностью задокументированы и валидированы. Но, по моему опыту, так бывает редко. В конечном счете системные требования должны быть полными, точными, корректными и согласованными; и чем раньше это будет достигнуто, тем лучше. Однако нередко именно на команду ПО ложится задача выявлять недостатки или неоднозначности системных требований и работать вместе с системной командой над формированием полностью валидированного набора системных требований.

На инженера по требованиям часто оказывается значительное давление с требованием создать спецификацию требований, не потратив предварительно время на анализ проблемы. Это приводит к требованиям, вымученным грубой силой: громоздким, неорганизованным и неясным. Подобно тому как художник тратит время на замысел шедевра, инженер по требованиям должен иметь время на сбор, анализ и организацию работ по требованиям. Речь идет о том, чтобы ясно представить себе проблему. Когда это сделано, собственно написание спецификации происходит относительно быстро и требует меньше переделок.

Инженеры по требованиям к программному обеспечению обычно выполняют следующие мероприятия по сбору и анализу, чтобы разработать требования к ПО.

6.6.1 Мероприятия по сбору требований

Прежде чем написать хотя бы одно требование, инженеры по требованиям собирают данные и знания, чтобы понять продукт, который они задают. Мероприятия по сбору включают следующее:

1. Проанализировать и стремиться как можно глубже понять системные требования и требования по безопасности в том виде, в каком они существуют. Инженеры по программному обеспечению должны очень хорошо знать системные требования. Понимание предварительной оценки безопасности также необходимо для осмысления факторов, определяющих требования безопасности.
2. Встречаться с заказчиками, системными инженерами и экспертами предметной области, чтобы получить ответы на любые вопросы по системным требованиям и восполнить недостающую информацию.
3. Определить зрелость и полноту системных требований и требований по безопасности до разработки требований к программному обеспечению.
4. Работать вместе с системными инженерами над внесением изменений в системные требования. Прежде чем команда программного обеспечения сможет уточнить системные

требования до уровня требований к ПО, системные требования должны быть относительно зрелыми и стабильными. Некоторые системные команды очень отзывчивы и тесно работают с командой ПО, обновляя системные требования. Однако во многих случаях команда ПО должна проактивно подталкивать системных инженеров к необходимым изменениям.

5. Учитывать соответствующие прошлые проекты, а также сообщения о проблемах по этим проектам. Нередко у заказчика, системных инженеров или разработчиков программного обеспечения уже есть некоторый прошлый опыт в данной предметной области. Требования в их прежнем виде могут быть непригодны для прямого использования, но они определенно могут помочь понять, что было реализовано, что сработало и, возможно, что не сработало.
6. Полностью изучить стандарты требований и ожидания в части сертификации.

6.6.2 Мероприятия по анализу требований

Процесс анализа подготавливает инженера к написанию требований. Иногда возникает соблазн сразу же броситься к написанию спецификации требований. Однако если предварительно не выполнить анализ и не рассмотреть проблему с нескольких точек зрения, в будущем это может привести к значительным переделкам. Некоторая итеративность и доводка будут всегда, но процесс анализа помогает свести их к минимуму. В ходе анализа требований следует учитывать следующее:

1. Организовать входные данные, полученные в ходе сбора, так, чтобы они были как можно более ясными и полными. Инженер по требованиям должен анализировать решаемую проблему с нескольких точек зрения. Нередко для рассмотрения проблемы с точки зрения пользователя используются варианты использования.[*]
2. Наметить каркас задачи по подготовке спецификации требований. Определив то, что, вероятно, станет оглавлением, можно организовать требования в логически выстроенный поток. Этот каркас также служит мерой полноты. Одна из распространенных стратегий определения каркаса требований состоит в том, чтобы перечислить функции безопасности и системные функции, которые должны быть обеспечены, а затем определить, какое программное обеспечение необходимо для работы каждой функции предусмотренным образом и для защиты от непредусмотренных эффектов.
3. Разрабатывать модели поведения программного обеспечения, чтобы обеспечить понимание потребностей заказчика. Некоторые из этих моделей будут уточняться и включаться в требования к ПО, а некоторые останутся лишь средством, помогающим разработке требований. Некоторые модели также могут быть добавлены в системные требования, когда обнаруживаются их недостатки.

4. В некоторых случаях для помощи в разработке требований может использоваться прототип. Аналитик требований может либо участвовать в разработке прототипа, либо использовать прототип для документирования требований. Быстрый прототип может быть очень полезен для демонстрации функциональности и уточнения деталей требований. Риск прототипа состоит в том, что он может выглядеть впечатляюще на поверхности, несмотря на плохую лежащую в основе архитектуру и код; поэтому руководители проекта или заказчики настаивают на использовании кода из “работающего” прототипа. Когда приходится сталкиваться с давлением по стоимости и срокам, я видела несколько проектов, которые отказались от своих исходных (и утвержденных) планов и попытались использовать код прототипа для летных испытаний и сертификации. Мне еще не доводилось видеть, чтобы это хорошо закончилось; страдают и стоимость, и сроки, и отношения с заказчиком. В разделе 6.10 приведены дополнительные соображения о прототипировании.

6.7 Написание требований к программному обеспечению

Мероприятия по сбору и анализу продолжают постоянно. Когда накоплено достаточное понимание и проблема в достаточной степени проанализирована, начинается собственно написание требований. Написание требований включает множество параллельных и итеративных мероприятий. Здесь они представлены как *задачи*, поскольку не являются последовательными мероприятиями. На следующих нескольких страницах объясняются шесть задач.

6.7.1 Задача 1: Определить методологию

Существует несколько подходов к документированию требований: от полностью текстового до полностью графического и до сочетания текста и графики. Из-за необходимости трассируемости и верифицируемости многие требования к программному обеспечению, критичному по безопасности, в основном являются текстовыми, а графика используется для дополнительной иллюстрации текста. Тем не менее графика играет значительную роль в разработке требований. «Изображения помогают преодолевать языковые барьеры и различия в словаре ...» [4]. На этой стадии графика должна быть сосредоточена на требованиях, то есть на том, *что* будет делать ПО, а не на проектировании, то есть на том, *как* ПО будет это делать. Многие графические представления могут быть дополнительно детализированы на этапе проектирования, но на этапе требований они должны по возможности оставаться свободными от реализации. Разработчики могут захотеть фиксировать некоторые проектные идеи в процессе написания требований, но это должны быть примечания для проектировщика, а не часть спецификации требований к ПО.

Небольшое предупреждение: не следует полагаться только на изображения, которые труд-

но испытывать. При использовании графики для описания требований нужно постоянно учитывать верифицируемость требований.

Некоторые примеры графических приемов, дополняющих текстовые описания, включают следующее [9]:

- *Контекстные диаграммы или диаграммы вариантов использования* – иллюстрируют интерфейсы с внешними сущностями. Детали интерфейсов обычно определяются в спецификации управления интерфейсами.
- *Высокоуровневый словарь данных* – описывает элементы данных и потоки данных между функциями и компонентами программного обеспечения. На этапе проектирования этот словарь обычно уточняется и детализируется.
- *Диаграммы сущность-связь или диаграммы классов* – показывают логические отношения между сущностями.
- *Диаграммы переходов состояний* – показывают переходы между состояниями внутри программного обеспечения. Каждое состояние обычно описывается в текстовом формате. *Диаграммы последовательностей* – показывают последовательность событий во время выполнения, а также временные характеристики этих событий.
- *Логические диаграммы и (или) таблицы решений* – определяют логические решения функциональных элементов.
- *Блок-схемы или диаграммы активности* – определяют пошаговый поток и решения. *Графические пользовательские интерфейсы* – проясняют связи, которые может быть трудно описать текстом.

Подход модельно-ориентированной разработки стремится увеличить графическое представление требований за счет использования моделей. Однако на сегодняшний день даже модели требуют некоторого текстового описания. Модельно-ориентированная разработка рассматривается в главе 14.

Выбранный прием должен учитывать связь между текстовыми и графическими представлениями. Текстовые требования обычно задают контекст и содержат ссылки на графические иллюстрации. Это помогает трассируемости и полноте, а также облегчает тестирование. Иногда графика приводится *только для справки* в поддержку требований. Если это так, это должно быть четко указано.

Еще один аспект методологии, который следует рассмотреть до того, как углубляться в написание требований – использовать ли средства автоматизированной разработки программного обеспечения (computer-aided software engineering, CASE) и шаблоны требований, поскольку они могут влиять на методологию.

В идеале применение выбранной методологии объясняется и иллюстрируется в стандартах требований. Стандарты помогают направлять авторов требований и обеспечивают, чтобы все следовали одному и тому же подходу. Если в написании требований участвуют несколько разработчиков, пример методологии и компоновки должен быть задокументирован, чтобы все применяли их одинаковым образом. Чем больше приведено примеров и деталей, тем лучше.

6.7.2 Задача 2: Определить компоновку документа требований высокого уровня к ПО (Software Requirements Document, SWRD)

Конечным результатом процесса документирования требований к программному обеспечению является документ требований высокого уровня к ПО. Этот документ определяет функции и возможности, которые должна обеспечивать программная система, и ограничения, которые она должна соблюдать. Документ требований высокого уровня к ПО является основой для всего последующего планирования проекта, проектирования и кодирования, а также фундаментом для системного тестирования и пользовательской документации. Он должен настолько полно, насколько это необходимо, описывать поведение [программной] системы в различных условиях. Он не должен содержать сведений о проектировании, построении, тестировании или управлении проектом, за исключением известных ограничений проектирования и реализации [4].[*]

Документ требований высокого уровня к ПО должен всесторонне объяснять функциональность и ограничения программного обеспечения; он не должен оставлять места для допущений. Если какая-либо функция или характеристика качества не отражена в этом документе, никто не должен ожидать, что она волшебным образом появится в конечном продукте [4].

На раннем этапе разработки требований следует определить общую компоновку документа требований высокого уровня к ПО. Позже она может быть изменена, но полезно иметь общее оглавление или каркас, с которого можно начать, особенно если участвуют несколько разработчиков. Оглавление или шаблон помогают всем сохранять фокус на целях своей области и понимать, что будет рассмотрено в других местах. Для повышения читаемости и удобства использования этого документа предлагаются следующие рекомендации:

- Включите оглавление с подразделами.
- Дайте обзор компоновки документа, включая краткое резюме каждого раздела и взаимосвязь между разделами.
- Определите ключевые термины, которые будут использоваться на протяжении всех требований, и применяйте их последовательно. Например, это могут быть системы координат и терминология для обозначения внешних функций. Если требования будут реализовываться или верифицироваться людьми, не знакомыми с языком или предмет-

ной областью, это критически важный элемент документа требований высокого уровня к ПО.

- Приведите полный и точный перечень сокращений.
- Объясните группировку требований в документе (для этого может оказаться полезной графика, например контекстная диаграмма).
- Логично организуйте документ и используйте обозначения и номера разделов и подразделов (обычно требования организуются по функциям или ключевой функциональности).
- Опишите среду, в которой будет работать программное обеспечение. Используйте по всему документу свободное пространство, чтобы повысить читаемость.
- Используйте полужирное начертание, курсив и подчеркивание для выделения и применяйте их последовательно по всему документу. Обязательно объясните значение любых соглашений, стилей текста и т.д.
- Выделяйте функциональные требования, нефункциональные или неповеденческие требования и внешние интерфейсы.
- Включайте все применимые ограничения (например, ограничения по инструментам, ограничения по языку, ограничения совместимости, аппаратные ограничения или соглашения по интерфейсам).
- Нумеруйте и маркируйте рисунки и таблицы и четко ссылайтесь на них из текстовых требований.
- При необходимости включайте перекрестные ссылки внутри спецификации и на другие данные.

6.7.3 Задача 3: Разделить функциональность программного обеспечения на подсистемы и (или) функции

Важно декомпозировать программное обеспечение на управляемые группы, которые имеют смысл и могут быть легко интегрированы. В большинстве случаев для представления высокоуровневой картины организации требований используется контекстная диаграмма или вариант использования.

Для более крупных систем программное обеспечение может быть разделено на подсистемы. Для меньших систем и подсистем ПО часто дополнительно разделяется на функции, причем каждая функция содержит конкретную функциональность.

Как отмечалось ранее, один из способов организовать функции состоит в том, чтобы совместно с инженерами по безопасности и системными инженерами определить функции

безопасности и системные функции, которые должны быть обеспечены. Затем этот вход используется для определения того, какое программное обеспечение необходимо для того, чтобы каждая функция работала предусмотренным образом, а также для определения того, какие меры защиты нужны для предотвращения непредусмотренных эффектов.

Существуют и другие способы организации требований. Независимо от выбранного подхода, при разделении функциональности обычно важно учитывать повторное использование и минимизацию влияния изменений.

6.7.4 Задача 4: Определить приоритеты требований

Поскольку проекты часто работают в сжатые сроки и разрабатываются с использованием итеративной или спиральной модели жизненного цикла, может потребоваться расставить приоритеты, какие требования должны быть определены и реализованы в первую очередь. Приоритеты следует согласовывать с системной командой и заказчиками. После базовых функций ПО, необходимых для загрузки, выполнения и ввода-вывода, наивысший приоритет должны иметь программные функции, критичные для функциональности системы или безопасности, либо обладающие высокой сложностью. Нередко приоритеты основываются на срочности, определенной заказчиком, и важности для общей функциональности системы. Чтобы корректно расставить приоритеты, иногда полезно разделить подсистемы, функции и (или) возможности на четыре группы:

1. срочно/важно (высокий приоритет);
2. не срочно/важно (средний приоритет);
3. срочно/не важно (низкий приоритет);
4. не срочно/не важно (может не потребоваться реализовывать).

Для крупных и сложных проектов при определении приоритета необходимо оценивать много факторов. В общем случае предпочтительно, чтобы процесс определения приоритетов оставался как можно более простым и свободным от политических игр. Приоритезация подсистем, функций и (или) возможностей должна быть указана в плане управления проектом. Помните, что по мере продвижения проекта приоритеты могут потребоваться скорректировать в зависимости от обратной связи заказчика и потребностей проекта.

6.7.5 Краткое отступление (не задача): опасные уклоны, которых следует избегать

Прежде чем обсуждать задачу документирования требований, давайте сделаем короткое отступление и рассмотрим несколько опасных уклонов, на которые, похоже, пытаются выйти многие неуспешные проекты. Они упоминаются здесь потому, что проекты иногда начинают двигаться по одному или нескольким из этих опасных уклонов вместо того, чтобы сосредоточиться на требованиях. Как только проект начинает скатываться по такому укло-

ну, выйти из него и снова сфокусироваться на требованиях становится трудно.

6.7.5.1 Опасный уклон No. 1: слишком быстрый переход к проектированию

Одна из самых трудных сторон написания требований состоит в том, чтобы не начать проектировать. По своей природе инженеры склонны решать задачи и хотят сразу перейти к проектированию. Давление сроков тоже вынуждает инженеров слишком быстро переходить к проектированию, еще до того, как они действительно зафиксировали, что именно пытаются построить. Однако слишком быстрый переход к проектированию может привести к компромиссной реализации, поскольку наилучший способ реализации часто неочевиден до тех пор, пока задача не определена полностью. Инженеры по требованиям могут рассматривать несколько потенциальных решений, чтобы подчеркнуть компромиссы между ними. Продумывание вариантов часто помогает повысить зрелость требований и выявить, что на самом деле необходимо программному обеспечению. Однако детали реализации не должны попадать в сами требования.

Чтобы избежать этого опасного уклона, я предлагаю инженерам делать следующее:

1. Четко определить в стандартах границу между требованиями (*что*) и проектированием (*как*).
2. Убедиться, что процесс работы с требованиями допускает примечания или комментарии. Такие аннотации дают ценную информацию, позволяющую передать замысел, не перегружая требования высокого уровня деталями проектирования.
3. Убедиться, что процесс допускает документирование проектных идей. Наличие некоторых намеченных потенциальных решений может дать хороший старт процессу проектирования.

6.7.5.2 Опасный уклон No. 2: один уровень требований

DO-178C указывает на необходимость как требований высокого уровня, так и требований низкого уровня. Требования высокого уровня документируются в документе требований высокого уровня к ПО, а требования низкого уровня являются частью проектирования. Разработчики нередко пытаются определить один уровень программных требований, по которому они будут непосредственно писать код, то есть объединяют требования высокого уровня и требования низкого уровня в один уровень требований. Хорошая инженерная практика и опыт постоянно предостерегают от этого. Требования и проектирование – это отдельные и различимые процессы. Из множества проектов, которые я оценивала, я лишь в редких случаях видела успешное объединение требований высокого уровня и требований низкого уровня в один уровень. Поэтому я рекомендую избегать такого подхода, за исключением хорошо обоснованных случаев. Я считаю более полезным документировать идеи реализации в черновом проектном документе во время написания требований, то есть вести работу

над требованиями и проектированием параллельно, а не объединять их. Некоторые полагают, что моделирование устранил необходимость в двух уровнях требований; однако, как будет обсуждаться в главе 14, это не так. Я предостерегаю всех, кто хочет оставить один уровень требований ради более быстрого перехода к реализации. Сначала это может выглядеть хорошо, но в итоге, как правило, заканчивается неудачей. Одной из областей, где возникают проблемы, является этап тестирования и верификации. Для более критичного программного обеспечения (уровни А и В) трудно добиться структурного покрытия, когда требования недостаточно детализированы (один уровень требований может не содержать достаточно деталей для полного структурного покрытия). Кроме того, для проектов уровня D, где тестирование требований низкого уровня не требуется, объединение требований может привести к увеличению объема тестирования, поскольку слияние требований высокого уровня и требований низкого уровня обычно дает более детализированные требования, чем традиционные требования высокого уровня.

вопрос 81 из раздела Frequently Asked Questions (FAQ) документа DO-248C под названием “Какие аспекты следует учитывать, когда существует только один уровень требований (или если требования высокого уровня и требования низкого уровня объединены)?” и документ CAST-15 группы органов сертификации по программному обеспечению (Certification Authorities Software Team, CAST) под названием “Merging High-Level and Low-Level Requirements” предостерегают от объединения программных требований в один уровень [10,11]. Есть несколько проектов, где требуется только один уровень программных требований (например, функциональность низкого уровня вроде частей операционной системы или функции математической библиотеки), но они составляют меньшинство.

6.7.5.3 Опасный уклон No. 3: сразу переходить к коду

За последние несколько лет усилилась тенденция сразу приступать к кодированию. У проекта могут быть некоторые концепции и несколько документированных требований, но когда на него давят, требуя как можно быстрее получить работающее программное обеспечение в эксплуатации, команда просто реализует все настолько хорошо, насколько может. Позже на разработчиков давят уже с тем, чтобы использовать этот наспех собранный или ситуативно написанный код в серийной версии продукта. Такой код обычно не соответствует требованиям, которые в конечном счете все же разрабатываются, в нем не задокументированы решения по проекту ПО и он часто не учитывает нештатные условия. Более того, прототипный код обычно создается грубой силой и не является наилучшим и самым безопасным решением. Я видела проекты, в которых к графику добавлялись месяцы и даже годы, пока команда пыталась восстановить проект ПО и недостающие требования по коду прототипа. Нередко в ходе этой обратной разработки проект обнаруживал, что код был неполным и неробастным, не говоря уже о его безопасности.

Теперь давайте выйдем с этих опасных уклонов и рассмотрим, что входит в документирование требований высокого уровня к программному обеспечению (надеюсь, это поможет командам избегать таких уклонов).

6.7.6 Задача 5: Задокументировать требования

Одна из трудностей написания требований состоит в определении нужного уровня детализации. Иногда системные требования очень подробны, и это вынуждает делать требования к программному обеспечению более детализованными, чем хотелось бы. В других случаях системные требования слишком расплывчаты и требуют дополнительной работы и декомпозиции со стороны авторов требований к ПО. Уровень детализации – это вопрос инженерного суждения; однако требования должны в достаточной мере объяснять, что будет делать ПО, но не углубляться в детали реализации. При написании требований высокого уровня к ПО важно помнить, что впереди еще слой проектирования, который включает требования низкого уровня к ПО. Требования высокого уровня к ПО должны давать достаточную детализацию для работ по проектированию, но не превращаться в проектирование.

6.7.6.1 Задокументировать функциональные требования

Большинство требований в документе требований высокого уровня к ПО являются функциональными требованиями (их также называют поведенческими требованиями). Функциональные требования точно определяют, какие входные данные ожидаются программным обеспечением, какие выходные данные будут формироваться ПО и каковы детали соотношений, существующих между этими входами и выходами. Иными словами, поведенческие требования описывают все аспекты интерфейсов между ПО и его средой (то есть аппаратурой, людьми и другим ПО) [12].

По сути, функциональные требования определяют, что делает программное обеспечение. Как обсуждалось ранее, они обычно организуются по подсистемам или функциям и документируются с использованием сочетания текста на естественном языке и графики.

При документировании функциональных требований следует учитывать следующие положения:

- Организуйте требования в логические группы.
- Стремитесь к тому, чтобы требования были понятны заказчикам и пользователям, не являющимся специалистами по программному обеспечению.
- Документируйте требования так, чтобы они были ясны проектировщикам (используйте комментарии или примечания для разъяснения потенциально сложных областей).
- Документируйте требования с акцентом на внешнее поведение программного обеспечения, а не на его внутреннее поведение (это следует оставить для проектирования).

- Документируйте требования так, чтобы их можно было тестировать.
- Используйте подход, допускающий модификации (включая нумерацию и организацию требований).
- Указывайте источник требований (см. разделы 6.7.6.6, 6.11, где подробнее рассматривается трассируемость).
- Последовательно используйте текст и графику.
- Идентифицируйте каждое требование (см. раздел 6.7.6.4 об уникальной идентификации). Сводите избыточность к минимуму. Вероятность расхождений возрастает каждый раз, когда требования формулируются заново.
- Следуйте стандартам требований, согласованным методам и шаблону. Если стандарт, метод или шаблон не отвечают конкретной потребности, определите, нужно ли обновить планы, стандарты или процедуры, либо оформить отступление от них.
- Если используется инструмент управления требованиями, следуйте согласованному формату и заблаговременно заполняйте все соответствующие поля (подробнее это обсуждается в разделе 6.9).
- Реализуйте характеристики хороших требований (см. раздел 6.7.6.9). Координируйтесь с коллегами, чтобы получить раннюю обратную связь и обеспечить согласованность в рамках команды.
- Выделяйте требования, связанные с безопасностью. Большинство компаний считают полезным выделять требования, которые напрямую вносят вклад в безопасность. Это требования, напрямую связанные с оценкой безопасности и поддерживающие соответствие ARP4754A. Документируйте производные требования и включайте причину их существования, то есть обоснование. (Производные требования будут обсуждаться позже.) Включайте и выделяйте требования робастности. Робастность – это “степень, в которой система продолжает функционировать надлежащим образом при столкновении с недопустимыми входными данными, дефектами в подключенных программных или аппаратных компонентах или неожиданными условиями эксплуатации” [4].

Для каждого требования следует рассматривать, существуют ли какие-либо потенциальные аномальные условия (например, недопустимые входные данные или недопустимые состояния), и обеспечивать наличие определенного поведения для каждого такого условия.

6.7.6.2 Задokumentировать нефункциональные требования

Нефункциональные (неповеденческие) требования – это требования, которые “определяют общие качества или атрибуты, которыми должно обладать результирующее программное обеспечение” [12]. Хотя они и не описывают функциональность, документировать эти тре-

бования важно, поскольку заказчик их ожидает и они влияют на проект ПО. Эти требования важны потому, что они объясняют, насколько хорошо будет работать продукт. К ним относятся такие характеристики, как скорость работы, удобство использования, интенсивность отказов и реакции на них, а также обработка аномальных условий [4]. По существу, нефункциональные требования включают ограничения, которые должны понимать проектировщики.

Когда нефункциональные требования различаются по функциям или подсистемам, их следует задавать вместе с соответствующей функцией или подсистемой. Когда нефункциональные требования распространяются на все функции или подсистемы, их обычно включают в отдельный раздел. Нефункциональные требования должны быть обозначены как таковые либо отдельным разделом требований, либо каким-либо атрибутом, поскольку обычно они не трассируются вниз к коду и влияют на стратегию тестирования. Нефункциональные требования все равно должны верифицироваться, но часто это делается анализом или инспекцией, а не тестированием, поскольку они могут не проявляться как функциональность, проверяемая тестированием.

Ниже приведены некоторые распространенные типы требований, которые относятся к нефункциональным:

1. *Требования к производительности* – вероятно, самый распространенный класс нефункциональных требований. Они включают информацию, полезную для проектировщиков, например время отклика, вычислительную точность, ожидания по времени, требования к памяти и пропускную способность.
2. *Требования безопасности*, которые могут не входить в состав функциональности, документируются как нефункциональные требования. Примеры включают следующее:
 - a. *Защита данных* – предотвращение потери данных или их повреждения.
 - b. *Нормативные требования по безопасности* – задание конкретных нормативных указаний или правил, которые должны быть выполнены.
 - c. *Доступность* – определение времени, в течение которого программное обеспечение будет доступно и полностью работоспособно.
 - d. *Надежность* – указание случаев, когда некоторый аспект программного обеспечения используется для поддержки надежности системы.
 - e. *Запасы по безопасности* – задание запасов или допусков, необходимых для обеспечения безопасности, например требований по запасу времени или памяти.
 - f. *Partitioning (обособление)* – обеспечение сохранения целостности обособления.

- g. *Деградация обслуживания* – объяснение того, как программное обеспечение будет плавно деградировать или действовать при наличии отказа.
 - h. *Робастность* – указание того, как программное обеспечение будет реагировать при наличии аномальных условий.
 - i. *Целостность* – защита данных от повреждения или неправильного выполнения.
 - j. *Латентность* – защита от скрытых отказов.
3. *Требования по защищенности* могут быть необходимы для поддержки безопасности, обеспечения надежности системы или защиты конфиденциальной информации.
 4. *Требования к эффективности* – это мера того, насколько хорошо система использует вычислительную мощность процессора, память или коммуникации [4]. Они тесно связаны с требованиями к производительности, но могут задавать и другие важные характеристики программного обеспечения.
 5. *Требования к удобству использования* определяют, какие характеристики нужны, чтобы программное обеспечение было удобным для пользователя; сюда относятся и аспекты человеческого фактора.
 6. *Требования к сопровождаемости* описывают необходимость легко модифицировать или исправлять программное обеспечение. Это включает сопровождаемость в ходе первоначальной разработки, во время интеграции и после ввода ПО в эксплуатацию.
 7. *Требования к переносимости* касаются необходимости легко переносить программное обеспечение в другие среды или на другие целевые вычислители.
 8. *Требования к повторному использованию* определяют необходимость использовать программное обеспечение в других приложениях или системах.
 9. *Требования к тестопригодности* описывают, какие возможности должны быть встроены в программное обеспечение для тестирования, включая системное или программное тестирование в разработке, интеграционное тестирование, тестирование у заказчика, тестирование на самолете и производственное тестирование.
 10. *Требования к интероперабельности* документируют, насколько хорошо программное обеспечение может обмениваться данными с другими компонентами. Могут применяться специальные стандарты интероперабельности.
 11. *Требования к гибкости* описывают необходимость легко добавлять новую функциональность в программное обеспечение как в ходе первоначальной разработки, так и на протяжении жизненного цикла продукта.

6.7.6.3 Задокументировать интерфейсы

Интерфейсы включают пользовательские интерфейсы (например, в системах отображения), аппаратные интерфейсы (например, протокол обмена для конкретного устройства), программные интерфейсы (такие как интерфейс прикладного программирования или библиотечный интерфейс) и коммуникационные интерфейсы (например, при использовании шины данных или сети).

Требования к интерфейсам с аппаратурой, программным обеспечением и базами данных должны быть задокументированы. Нередко документ требований высокого уровня к ПО ссылается на документ управления интерфейсами. В некоторых случаях данные и управляющие интерфейсы описываются явными требованиями этого документа, независимыми стандартами или словарем данных. Интерфейсы должны быть задокументированы таким образом, чтобы поддерживать анализ межкомпонентной связности по данным и по управлению, который рассматривается в главе 9.

Любые документы по интерфейсам, на которые ссылаются требования, должны находиться под управлением конфигурацией, поскольку они влияют на требования, тестирование, эксплуатацию системы и сопровождение программного обеспечения.

6.7.6.4 Уникально идентифицировать каждое требование

Каждое требование должно иметь уникальную метку (ее также называют номером, ярлыком или идентификатором). Большинство организаций используют слово *должен (shall)* для идентификации требований. Каждое вхождение слова *должен (shall)* обозначает одно требование и имеет свою метку. Такой подход помогает инженеру по требованиям отличать то, что действительно требуется, от комментариев или вспомогательной информации.

Некоторые инструменты автоматически присваивают требованиям метки, тогда как другие допускают ручное присвоение. Подход к идентификации должен быть задокументирован в стандартах и строго соблюдаться. После того как метка была использована, ее не следует присваивать повторно, даже если требование удалено. Кроме того, важно следить, чтобы каждой метке соответствовало только одно требование. Иными словами, не следует объединять несколько требований в одно, поскольку это приводит к неоднозначности и затрудняет подтверждение полноты тестирования.

6.7.6.5 Задокументировать обоснование

Хорошей практикой является включение обоснования вместе с требованиями, поскольку это может повысить качество требования, сократить время, необходимое для его понимания, повысить точность, сократить время на сопровождение и быть полезным для обучения инженеров функциональности программного обеспечения. Сам процесс написания обоснования не только улучшает понимание требований читателем, но и помогает авторам писать

более качественные требования. В *Requirements Engineering Management Handbook* Федерального управления гражданской авиации США говорится:

Выработать обоснование для плохого требования или допущения может быть трудно. Требование к разработчику спецификации подумать, почему данное требование необходимо или почему делается данное допущение, часто повышает качество требования... Требования документируют, что система будет делать. Проектирование документирует, как система будет это делать. Обоснование документирует, почему требование существует или почему оно сформулировано именно так. Обоснование следует включать всякий раз, когда в требовании есть что-то, что может быть неочевидно читателю или что может помочь читателю понять, почему данное требование существует [13].

Далее в руководстве приводятся рекомендации по написанию обоснования, включая следующие [13]:

- Предоставляйте обоснование на протяжении всей разработки требований, чтобы объяснять, почему требование необходимо и почему включены конкретные значения.
- Избегайте задания требований в обосновании. Если информация в обосновании существенна для требуемого поведения системы, она должна быть частью требований, а не обоснования.
- Предоставляйте обоснование, когда причина существования требования неочевидна.
- Включайте обоснование для допущений об окружающей среде, от которых зависит система.
- Предоставляйте обоснование для значений и диапазонов в каждом требовании.
- Держите каждое обоснование коротким и относящимся к объясняемому требованию. Фиксируйте обоснование как можно раньше, чтобы не потерять ход мысли.

6.7.6.6 Трассировать требования к их источнику

Каждое требование должно трассироваться к одному или нескольким родительским требованиям, то есть к требованиям более высокого уровня, из которых оно было декомпозировано. Инженеры по требованиям должны обеспечивать, чтобы каждое системное требование, распределенное на программное обеспечение, было полностью реализовано требованиями к ПО, которые к нему трассируются, то есть его дочерними требованиями.

Трассируемость должна документироваться по мере написания требований. Практически невозможно вернуться позже и корректно исправить трассировку. Многие инструменты управления требованиями позволяют включать данные трассировки, но сами инструменты не знают автоматически, каковы отношения трассировки. Разработчики должны дисциплинированно трассировать требования по мере их разработки.

Помимо трассировки вверх, требования должны быть написаны таким образом, чтобы их можно было трассировать вниз к требованиям низкого уровня и тестовым примерам. Дополнительная информация о трассируемости приведена в разделе 6.11.

6.7.6.7 Выявлять неопределенности и допущения

Во время определения требований часто бывает неизвестна часть информации. Такие элементы можно пометить как *TBD* (to be determined, подлежит определению) или иной ясной нотацией. Хорошей практикой является включение примечания или сноски, указывающей, кто отвечает за закрытие данного *TBD* и когда это будет сделано. Все *TBD* должны быть закрыты до формального рассмотрения требований и до начала реализации. Аналогично, любые *допущения* должны быть задокументированы, чтобы соответствующие команды (например, системная, аппаратная, верификационная или по безопасности) могли их подтвердить и верифицировать.

6.7.6.8 Начать словарь данных

Некоторые проекты пытаются обходиться без словаря данных, потому что его сопровождение может быть утомительным. Однако для систем, интенсивно работающих с данными, словарь данных чрезвычайно ценен. Большая часть словаря данных заполняется во время проектирования. Тем не менее полезно начинать документировать общие данные (включая смысл данных, тип, длину, формат и т.д.) уже на этапе определения требований. Словарь данных помогает при интеграции и общей согласованности. Он также помогает предотвращать ошибки, вызванные неодинаковым пониманием данных [4]. В зависимости от особенностей проекта словарь данных и документ управления интерфейсами могут быть объединены.

6.7.6.9 Реализовать характеристики хороших требований

В главе 2 были определены характеристики хороших системных требований. Требования к программному обеспечению должны обладать теми же характеристиками, а именно быть атомарными, полными, краткими, согласованными, корректными, свободными от деталей реализации, необходимыми, трассируемыми, однозначными, верифицируемыми и реализуемыми. Ниже приведены дополнительные рекомендации по написанию высококачественных требований к ПО:

- Используйте краткие и полные предложения с правильной грамматикой и орфографией. Используйте одно вхождение слова *должен (shall)* для каждого требования.
- Используйте действительный залог.
- Подчеркивайте важные элементы с помощью графики, полужирного выделения, последовательности, свободного пространства или другого метода.

- Используйте термины последовательно, как это определено в глоссарии документа требований высокого уровня к ПО или разделе определений.
- Избегайте неоднозначных терминов. Примеры неоднозначных формулировок: *как целевой ориентир, настолько, насколько это практически возможно, модульный, достижимый, достаточный, своевременный, удобный для пользователя* и тому подобное. Если такие формулировки используются, их нужно количественно определять. Пишите требования на надлежащем уровне детализации. Обычно надлежащий уровень — это такой, который можно проверить одним или всего несколькими тестами. Сохраняйте требования на согласованном уровне детализации. Сводите к минимуму или избегайте использования слов, указывающих на наличие нескольких требований, таких как *если не* или *за исключением*.
- Избегайте использования *и/или* или символа косой черты (/) для разделения двух слов, поскольку это может быть неоднозначно.
- Осторожно используйте местоимения, например *оно* или *они*. Обычно лучше повторить существительное.
- Избегайте латинских сокращений *i.e.* («то есть») и *e.g.* («например»), поскольку их значения часто путают.
- Включайте обоснование и справочные сведения по требованиям в поле примечаний или комментариев (см. раздел 6.7.6.5). Нет ничего лучше хорошего комментария, чтобы понять, что именно было в голове у автора.
- Избегайте отрицательных требований, поскольку их трудно верифицировать.
- Обеспечивайте, чтобы требования полностью определяли функциональность, путем поиска пропусков, то есть того, что не задано, хотя должно быть задано.
- Встраивайте робастность в требования, продумывая, как программное обеспечение будет реагировать на аномальные входные данные.
- Избегайте слов, сходных по звучанию или написанию.
- Осторожно используйте оценочные наречия, например *разумно, быстро, существенно* и *периодически*, поскольку они могут быть неоднозначны.

Левесон подчеркивает важность полноты требований, когда пишет:

Наиболее важным свойством спецификации требований с точки зрения безопасности является полнота или отсутствие неоднозначности. Желаемое поведение программного обеспечения должно быть задано с достаточной степенью детализации, чтобы отличать его от любой нежелательной программы, которая могла бы быть разработана. Если документ тре-

бований содержит недостаточно информации для того, чтобы проектировщики могли различить наблюдаемо различные шаблоны поведения, представляющие желательное и нежелательное (или безопасное и небезопасное) поведение, то спецификация является неоднозначной или неполной [1].

6.7.7 Задача 6: Предоставить обратную связь по системным требованиям

Написание требований к программному обеспечению предполагает внимательное изучение системных требований. Команда ПО нередко обнаруживает ошибочные, отсутствующие или противоречащие друг другу системные требования. Любые выявленные проблемы в системных требованиях следует документировать в сообщении о проблеме, доводить до команды системной разработки и отслеживать до подтверждения того, что необходимые действия действительно выполнены. Во многих программах команда ПО исходит из того, что команда системной разработки исправила системные требования на основе устной обратной связи или сообщений электронной почты; однако в финальной, самой напряженной фазе сертификации выясняется, что системные требования так и не были обновлены. Чтобы избежать этой проблемы, команда ПО должна действовать проактивно и обеспечивать обновление системных требований, оформляя сообщения о проблемах по системным требованиям и отслеживая каждое такое сообщение о проблеме. Отсутствие должного сопровождения по найденным проблемам может привести к несоответствию между системными требованиями и функциональностью ПО, что способно затормозить процесс сертификации, поскольку такой разрыв в требованиях считается несоответствием DO-178C.

6.8 Верификация (рассмотрение) требований

Когда требования становятся зрелыми и стабильными, их верифицируют. Обычно это происходит путем проведения одного или нескольких экспертных рассмотрений. Цель процесса рассмотрения состоит в том, чтобы выявить ошибки до их реализации; поэтому это одно из наиболее важных и ценных мероприятий при разработке программного обеспечения, критичного по безопасности. При правильном выполнении рассмотрения позволяют предотвращать ошибки, экономить значительное время и снижать затраты. Экспертное рассмотрение выполняется группой из одного или нескольких рецензентов.

Для оптимизации процесса рассмотрения я рекомендую две стадии экспертных рассмотрений: неформальную и формальную. Сначала идет неформальная стадия, которая помогает как можно быстрее довести требования до большей зрелости. Более того, как уже отмечалось ранее, я поддерживаю концепцию командной разработки, когда по меньшей мере два разработчика совместно вырабатывают требования, непрерывно консультируются друг с другом и взаимно проверяют работу друг друга. Цель состоит в том, чтобы проводить частые неформальные рассмотрения на ранних этапах и тем самым минимизировать коли-

чество проблем, обнаруживаемых в ходе формального экспертного рассмотрения.

Во время формального рассмотрения требований рецензенты используют контрольный перечень (который обычно включается в План верификации ПО (Software Verification Plan, SVP) или в стандарты требований). Исходя из DO-178C, в такой контрольный перечень обычно, как минимум, включаются и в ходе рассмотрения оцениваются следующие пункты [3]:[*]

- Выполнены входные критерии, определенные в планах для данного рассмотрения.[†]
В большинстве случаев для этого нужны выпущенные системные требования, выпущенные стандарты требований к ПО, выпущенные План разработки ПО (Software Development Plan, SDP) и План верификации ПО, а также введенный контроль конфигурации требований к ПО.
- Требования высокого уровня к ПО соответствуют системным требованиям. Это гарантирует, что требования высокого уровня полностью реализуют те системные требования, которые распределены на ПО.
- Требования высокого уровня к ПО трассируются к системным требованиям. Это двуправленная трассировка: все системные требования, распределенные на ПО, должны иметь реализующие их требования высокого уровня, а все требования высокого уровня (за исключением производных требований) должны трассироваться к системным требованиям. Дополнительные замечания по трассируемости приведены в разделе 6.11.
- Требования высокого уровня к ПО являются точными, однозначными, согласованными и полными. Это включает обеспечение того, чтобы входы и выходы были четко определены и представлены в количественной форме (включая единицы измерения, диапазон, масштабирование, точность и частоту поступления), чтобы были рассмотрены как нормальные, так и аномальные условия, чтобы все диаграммы были точными и ясно промаркированы, и т.д.
- Требования высокого уровня к ПО соответствуют стандартам требований к ПО. Как рекомендовалось ранее, такие стандарты должны включать перечисленные выше признаки хороших требований. Стандарты требований рассматривались в главе 5.
- Требования высокого уровня к ПО являются верифицируемыми; например, требования, включающие измерения, содержат допуски, на один идентификатор или метку приходится только одно требование, используются количественно определенные термины и отсутствуют отрицательные требования.
- Требования высокого уровня к ПО однозначно идентифицированы. Как уже отмечалось, каждое требование должно содержать слово *должен (shall)* и уникальный иден-

тификатор или метку.

- Требования высокого уровня к ПО совместимы с целевым вычислителем. Цель состоит в том, чтобы убедиться, что требования высокого уровня согласуются с аппаратными и программными особенностями целевого вычислителя, особенно в части времен отклика и аппаратуры ввода-вывода. Нередко этот пункт лучше проверяется при рассмотрении проекта ПО, чем при рассмотрении требований высокого уровня.
- Предлагаемые алгоритмы, особенно в области разрывов непрерывности, проанализированы, чтобы подтвердить их корректность и характер поведения.
- Производные требования являются уместными, должным образом обоснованы и переданы команде по безопасности.
- Для каждого режима работы задокументированы функциональные и эксплуатационные требования.
- В требованиях высокого уровня должны быть отражены критерии производительности, например прецизионность и точность.
- В требованиях высокого уровня также указывают требования и ограничения по времени, ограничения по объему памяти, аппаратные и программные интерфейсы (например, протокол, форматы и частоту входных и выходных данных), а также требования к обнаружению отказов и мониторингу безопасности.
- Если применяется обособление, требования высокого уровня должны задавать соответствующие требования: как программные компоненты взаимодействуют друг с другом и какой уровень ПО назначен каждому разделу (partition).

Во время формальных рассмотрений требования могут верифицироваться как единое целое либо как функциональные группы. Если рассмотрения разделены по функциональности, все равно должно существовать рассмотрение, охватывающее объединенные группы на предмет согласованности и целостности.

Чтобы рассмотрение было эффективным, необходимо собрать правильных рецензентов. В состав рецензентов должен входить квалифицированный технический персонал, включая разработчиков программного обеспечения, системных инженеров, инженеров по испытаниям, персонал по безопасности, специалиста по гарантии качества ПО (software quality assurance, SQA) и персонал по взаимодействию с сертифицирующим органом.[*] Крайне важно, чтобы каждый рецензент прочитал и полностью понял пункты контрольного перечня до проведения рассмотрения. Если рецензенты не знакомы с контрольным перечнем, следует провести обучение с пояснениями и примерами по каждому пункту перечня.

Замечания по итогам формального рассмотрения должны документироваться, классифи-

цироваться (например, существенная проблема, несущественная проблема, редакционное замечание, дублирующее замечание, без изменений) и получать решение. Автор замечания должен согласиться с предпринятым действием до закрытия рассмотрения. Контрольный перечень требований также должен быть успешно завершен до закрытия рассмотрения. Дополнительные рекомендации по процессу экспертного рассмотрения приведены ниже.

6.8.1 Рекомендуемые практики экспертного рассмотрения

Хотя это и не является обязательным требованием, большинство проектов используют формальный процесс экспертного рассмотрения для верификации своих планов, требований, проекта ПО, кода, тестовых примеров и тестовых процедур, результатов верификации ПО (Software Verification Results, SVR), указателей конфигурации ПО (Software Configuration Index, SCI), итоговых отчетов разработки ПО (Software Accomplishment Summary, SAS) и других ключевых данных ЖЦ ПО. Команда с правильно подобранным составом часто может обнаружить ошибки, которые отдельный человек пропустил бы. Один и тот же процесс экспертного рассмотрения может использоваться для различных наборов данных ЖЦ ПО (он не ограничивается требованиями). Иными словами, сам процесс рассмотрения может быть стандартизован, так что для конкретного рассмотрения будут меняться только контрольные перечни, проверяемые данные и состав рецензентов. В данном разделе перечислены некоторые рекомендуемые практики, которые следует встроить в процесс экспертного рассмотрения:

- Назначайте модератора или технического руководителя, который будет планировать рассмотрение, предоставлять данные, распределять задачи, собирать и консолидировать замечания, вести совещание по рассмотрению, обеспечивать заполнение контрольного перечня рассмотрения, следить за тем, чтобы все замечания были обработаны до закрытия рассмотрения, и т.д.
- Обеспечивайте, чтобы проверяемые данные находились под управлением конфигурацией. Это может быть неформальное или инженерное управление конфигурацией, но данные должны быть контролируемыми, а версия должна быть указана в записях экспертного рассмотрения. Идентифицируйте проверяемые данные и соответствующие версии в записи экспертного рассмотрения.
- Указывайте дату экспертного рассмотрения, приглашенных рецензентов, рецензентов, представивших замечания, и количество времени, которое каждый рецензент потратил на рассмотрение. Информация о рецензентах важна как свидетельство независимости и добросовестного выполнения обязанностей.
- При необходимости или если этого требует контракт либо процедуры, привлекайте заказчика. Используйте для рассмотрения контрольный перечень и убеждайтесь, что

все рецензенты обучены работе с этим перечнем и соответствующими стандартами для проверяемых данных. Контрольные перечни обычно включаются в утвержденные планы или стандарты либо на них даются ссылки.

- Предоставляйте рецензентам пакет рассмотрения (включая оцениваемые данные с номерами строк или разделов, контрольный перечень, форму рассмотрения и любые иные данные, необходимые для рассмотрения), а также определяйте обязательных и необязательных рецензентов. Распределяйте обязанности каждого рецензента (например, один человек проверяет трассируемость, другой – соответствие стандартам и т.д.). Обеспечивайте, чтобы обязательные рецензенты покрывали все аспекты контрольного перечня, чтобы рецензенты действительно выполняли свои задачи и чтобы они были квалифицированы для порученной работы. Если обязательный рецензент отсутствует или не может выполнить свою задачу, может потребоваться, чтобы рассмотрение выполнил кто-то другой с равной квалификацией, либо само рассмотрение придется перенести.
- Предоставляйте рецензентам инструкции (например, указывайте сроки представления замечаний, даты совещаний, роли, области фокуса, открытые вопросы, расположение файлов, справочные документы).
- Давайте рецензентам достаточное заблаговременное уведомление и время для выполнения рассмотрения. Если обязательному рецензенту требуется больше времени, перенесите рассмотрение.
- Обеспечивайте достижение требуемого уровня независимости. Таблицы приложения А в DO-178С указывают, при каком уровне программного обеспечения требуется независимость. В главе 10 обсуждается независимость верификации.
- Используйте квалифицированных рецензентов. Ключевые технические рецензенты включают тех, кто будет использовать данные (например, тестировщика и проектировщика), а также одного или нескольких независимых разработчиков (когда требуется независимость). Как отмечалось ранее, для рассмотрения требований рекомендуется привлекать системный персонал и персонал по безопасности. Качество рассмотрения будет не выше качества людей, которые его выполняют, поэтому в течение всего жизненного цикла проекта оправданно назначать на технические роли лучших и наиболее квалифицированных специалистов. Младшие инженеры могут учиться, выполняя вспомогательные роли.
- Приглашайте персонал по гарантии качества ПО и взаимодействию с сертифицирующим органом, а также любой другой вспомогательный персонал, который необходим.
- Поддерживайте разумный размер команды. Это субъективный вопрос, и он обычно

варьируется в зависимости от уровня программного обеспечения и значимости проверяемых данных.

- Обеспечьте способ документирования замечаний рецензентов (обычно используется электронная таблица или инструмент для комментариев). Рецензент обычно вносит следующие данные: имя рецензента, идентификатор и версию документа, номер раздела или строки документа, номер замечания, само замечание и классификацию замечания (существенное, несущественное, редакционное и т.д.).
- Планируйте совещание для обсуждения нетривиальных замечаний или вопросов. Некоторые компании предпочитают ограничивать количество совещаний и использовать их только для спорных тем. Если совещания не проводятся, необходимо обеспечить способ получения согласия всех рецензентов по требуемым действиям. По моему опыту, краткое совещание для обсуждения технических вопросов гораздо эффективнее и экономичнее, чем бесконечные цепочки электронных писем.
- Если предполагается проведение совещания, предоставьте автору данных время, чтобы до совещания рассмотреть и предложить ответ по каждому замечанию. Время совещания должно быть сосредоточено на нетривиальных вопросах, требующих очного взаимодействия. Если команда географически распределена, используйте средства электронного взаимодействия (например, WebEx, NetMeeting или Live Meeting) и телефонные конференции, чтобы привлечь нужных людей.
- Ограничивайте время обсуждения вопросов. Некоторые компании вводят правило 2 минут: любые обсуждения, требующие более 2 минут, откладываются для последующего рассмотрения. Если вопрос не может быть решен в отведенное время, организуйте последующее совещание с нужными заинтересованными сторонами. Модератор помогает удерживать обсуждения в рамках темы и графика.
- Определите процесс обсуждения спорных вопросов, например путь эскалации, ведущего арбитра или совет по контролю продукта.
- Заполняйте контрольный перечень рассмотрения. Возможны несколько подходов: контрольный перечень (или перечни) может заполняться каждым членом команды по своей части, командой в ходе совещания по экспертному рассмотрению либо квалифицированным рецензентом. Обычно невозможно корректно заполнить контрольный перечень до тех пор, пока не будут обработаны все замечания по рассмотрению.
- Убедитесь, что все вопросы обработаны и закрыты до закрытия рассмотрения. Если вопрос необходимо обработать, но это нельзя сделать до закрытия рассмотрения, следует оформить сообщение о проблеме. Номер сообщения о проблеме должен быть включен

в записи экспертного рассмотрения, чтобы гарантировать, что вопрос в конечном итоге будет обработан или должным образом разрешен.

- Разбивайте большие документы на меньшие пакеты и сначала рассматривайте области высокого риска. После того как все отдельные пакеты будут рассмотрены, опытный инженер или команда должны провести интеграционное рассмотрение, чтобы убедиться, что все пакеты согласованы между собой и корректны. Иными словами, рассматривайте данные совместно. Обеспечьте организованный способ хранения и извлечения записей рассмотрений и контрольных перечней, поскольку они являются свидетельством для сертификации.

Ниже приведены некоторые распространенные проблемы, возникающие при фактической реализации процесса экспертного рассмотрения, которых можно избежать при надлежащем управлении экспертными рассмотрениями:

- Рассмотрение этого мероприятия как *галочки*, а не как инструмента раннего разрешения технических вопросов, повышения ценности конечного продукта и экономии времени и денег на протяжении всего проекта.
- Непредоставление рецензентам времени для тщательного рассмотрения данных. Формирование чрезмерно большой команды рассмотрения.
- Использование неквалифицированных и плохо обученных рецензентов.
- Незакрытие замечаний до перехода к следующему этапу. Неполное или несвоевременное заполнение обязательного контрольного перечня.

6.9 Управление требованиями

6.9.1 Основы управления требованиями

Жизненно важной частью разработки программного обеспечения является управление требованиями. Как бы тщательно ни было выполнено планирование и как бы добросовестно ни были написаны требования, изменения все равно произойдут. Организованный процесс управления требованиями необходим для управления неизбежными изменениями. Управление требованиями включает “все мероприятия по поддержанию целостности, точности и актуальности согласованного набора требований по мере продвижения проекта” [4].

Для управления требованиями следует выполнять следующее:

- *Разрабатывайте требования так, чтобы их можно было модифицировать.* Как упоминалось ранее, модифицируемость является характеристикой хороших требований. Модифицируемые требования хорошо организованы, имеют надлежащий уровень детализации, свободны от деталей реализации, четко идентифицированы и трассируемы.

- *Устанавливайте базовую версию требований.* Как функциональные, так и нефункциональные требования должны быть включены в базовую версию. Обычно базовая версия устанавливается после того, как требования прошли экспертное рассмотрение. Для крупных проектов полезно иметь управление версиями как отдельных требований, так и разделов либо всего документа требований высокого уровня к ПО целиком.
- *Управляйте всеми изменениями базовой версии.* Обычно это происходит через процесс регистрации сообщений о проблемах и совет по управлению изменениями. Изменения требований выявляются, одобряются советом по управлению изменениями, реализуются и повторно рассматриваются.
- *Обновляйте требования с использованием утвержденного процесса.* Обновления требований должны выполняться с использованием того же процесса работы с требованиями, который определен в планах. Иными словами, следует соблюдать стандарты, реализовывать атрибуты качества, проводить рассмотрения и т.д. Некоторые компании добросовестно соблюдают процесс в первый раз, но становятся менее дисциплинированными при обновлениях. Из-за этой тенденции внешние аудиторы и инженеры по обеспечению качества обычно внимательно оценивают тщательность внесения изменений.
- *Повторно рассматривайте требования.* После внедрения изменений сами изменения и требования, затронутые этими изменениями, должны быть повторно рассмотрены. Если изменено несколько требований, может быть уместно командное рассмотрение. Если число измененных или затронутых требований невелико и изменения просты, рассмотрение может быть выполнено одним человеком. При этом все равно требуется надлежащий уровень независимости.
- *Отслеживайте состояние.* Состояние каждого требования должно отслеживаться. Типичными состояниями изменений требований являются: предложено, одобрено, реализовано, верифицировано, удалено или отклонено [9]. Нередко состояние управляется через процесс регистрации сообщений о проблемах, чтобы не иметь две параллельные системы статусов. Процесс регистрации сообщений о проблемах обычно включает следующие состояния: “предложено” (изменение требования предложено), “в работе” (изменение одобрено), “реализовано” (изменение внесено), “проверено” (изменение рассмотрено), “отклонено” (изменение не одобрено) и “закрыто” (изменение полностью реализовано, рассмотрено и находится под управлением конфигурацией).

Управление изменениями далее рассматривается в главе 10.

6.9.2 Инструменты управления требованиями

Большинство компаний используют коммерчески доступный инструмент управления требованиями для документирования и помощи в управлении своими требованиями; однако у некоторых есть собственный самодельный инструмент. Заказчики могут предписывать использование конкретного инструмента, чтобы обеспечить единообразный подход к управлению требованиями на всех иерархических уровнях.

Какой бы инструмент управления требованиями ни был выбран, он должен как минимум обеспечивать возможность выполнять следующее:

- Легко добавлять атрибуты или поля требований.
- Экспортировать данные в читаемый формат документа.
- Поддерживать графику (например, таблицы, блок-схемы, графику пользовательского интерфейса).
- Устанавливать базовую версию требований.
- Добавлять или удалять требования.
- Поддерживать работу нескольких пользователей в нескольких географических точках.
- Документировать комментарии или обоснование требований.
- Выполнять трассировку вверх и вниз.
- Генерировать отчеты по трассируемости.
- Защищать от несанкционированного изменения (например, паролем).
- Поддерживать резервное копирование.
- Переупорядочивать требования без их перенумерации.
- Управлять несколькими уровнями требований (например, системными, требованиями высокого уровня к ПО и требованиями низкого уровня к ПО).
- Добавлять уровни требований при необходимости.
- Поддерживать несколько программ разработки.

Типичные поля или атрибуты требований, включаемые в инструмент управления требованиями, таковы:

- *Идентификация требования* – уникальный идентификатор или метка требования.
- *Применимость требования* – если задействовано несколько проектов, некоторые требования могут быть применимы или неприменимы.
- *Описание требования* – формулирует требование.

- *Комментарий к требованию* – объясняет важные аспекты требования, такие как обоснование и связанные требования. Для производных требований сюда входит обоснование того, почему это производное требование необходимо.
- *Статус* – для идентификации состояния каждого требования (например, одобрено советом по управлению изменениями, в работе, реализовано, верифицировано, удалено или отклонено).
- *Полномочие на изменение* – идентифицирует сообщение о проблеме, номер запроса на изменение и т.д., использованные для санкционирования реализации или изменения требования.
- *Данные трассировки* – документируют трассировку вверх к родительским требованиям, вниз к дочерним требованиям и наружу к тестовым примерам и (или) тестовым процедурам.
- *Специальные поля* – идентифицируют требования по безопасности, производные требования, требования по робастности, статус одобрения и т.д.

Документ или электронная таблица могут использоваться для документирования требований. Однако чем сложнее проект и больше команда разработки, тем больше пользы приносит использование инструмента управления требованиями.

Если используется инструмент управления требованиями, важно иметь следующее:

- Техническую поддержку и обучение. Разработчики должны быть обучены правильному использованию инструмента.
- Проектно-специфические инструкции, объясняющие, как правильно использовать инструмент в данной среде (такая информация часто включается в стандарты или руководство по процессам).
- Часто задаваемые вопросы и примеры, помогающие решать типовые проблемы, которые будут встречаться.

Инструменты управления требованиями могут быть очень мощными. Они могут помогать управлять требованиями и версиями, поддерживать большие команды, облегчать трассируемость, отслеживать статус требований, поддерживать использование требований в нескольких проектах и многое другое. Однако даже при наличии инструмента управления требованиями все равно требуется хорошее управление изменениями требований. Инструмент не будет “компенсировать отсутствие процесса, дисциплины, опыта или понимания” [4].

6.10 Прототипирование требований

Алан Дэвис объясняет: “Прототипирование – это метод построения частичной реализации системы, чтобы заказчики, пользователи или разработчики могли лучше понять проблему или решение этой проблемы” [12]. Прототипы могут помогать получать обратную связь заказчика по ключевой функциональности, исследовать варианты проектирования для определения осуществимости, повышать зрелость требований, выявлять неоднозначные или неполные требования, минимизировать недопонимание требований и повышать робастность требований.

Однако, когда я слышу слово *прототип*, я обычно внутренне морщусь (по крайней мере немного). Многие проекты не только используют прототип для помощи в разработке требований, но и пытаются затем спасти код прототипа. Это редко бывает успешным. Код прототипа часто разрабатывается быстро, чтобы доказать концепцию. Обычно он не спроектирован робастно и не соответствует стандартам разработки. Прототипирование может быть отличным способом повысить зрелость требований, получить обратную связь заказчика и понять, что работает, а что нет. Проблема возникает тогда, когда заказчик или руководство хотят использовать код прототипа для интеграции и сертификации. Это может привести к обратной разработке проекта и требований и к значительной переработке кода. В итоге это занимает больше времени и создает больше проблем в коде, чем отказ от этого кода и начало работы заново.

Тем не менее исключения бывают, и прототипирование может успешно применяться, особенно когда код прототипа планируется заранее и является частью организованного процесса, а не просто запоздалой идеей, призванной наверстать потерянное время. Существует два распространенных подхода к успешному прототипированию [4,12]:

1. *Отбрасываемый прототип*. Это быстрый прототип, создаваемый без твердо зафиксированных требований и проекта, чтобы определить функциональную осуществимость, получить обратную связь заказчика и выявить отсутствующие требования. Отбрасываемый прототип используется для повышения зрелости требований, после чего код выбрасывается. Чтобы избежать искушения оставить прототип, следует:
2. реализовывать только часть функциональности;
3. заранее заключить твердое соглашение о том, что код будет выброшен;
4. использовать другую среду.

Без этих мер некоторые заказчики или руководители проекта будут склонны попытаться использовать прототип для сертификации. Отбрасываемые прототипы не создаются ни робастными, ни эффективными и не проектируются с расчетом на сопровождаемость. Сохра-

нять отбрасываемый прототип – большая ошибка.

2. *Эволюционный прототип*. Этот подход полностью отличается от предыдущего. Он разрабатывается с намерением использовать код и сопутствующие данные позднее. Эволюционный прототип обычно представляет собой частичную реализацию ключевой функциональности. После того как функциональность доказана, код приводится в порядок и добавляется дополнительная функциональность. Эволюционный прототип предназначен для использования в конечном продукте; поэтому для него применяется строгий процесс, учитываются качественные атрибуты требований и проекта, оцениваются несколько вариантов проектирования, реализуется робастность, соблюдаются стандарты требований и проекта, а также используются комментарии в коде и стандарты кодирования. Прототип образует основу конечного продукта и тесно связан со спиральной моделью жизненного цикла.

Оба подхода к прототипированию могут использоваться, но они должны быть описаны в планах, согласованы с сертифицирующим органом и реализованы именно так, как было согласовано.

6.11 Трассируемость

DO-178C требует двунаправленной трассируемости между системными требованиями и требованиями высокого уровня к ПО, между требованиями высокого уровня к ПО и требованиями низкого уровня к ПО, между требованиями низкого уровня к ПО и кодом, между требованиями и тестовыми примерами, между тестовыми примерами и тестовыми процедурами, а также между тестовыми процедурами и результатами тестирования [3]. В этом разделе рассматриваются:

1. важность и преимущества трассируемости;
2. нисходящая и восходящая трассируемость;
3. что DO-178C говорит о трассируемости;
4. каких трудностей трассируемости следует избегать.

6.11.1 Важность и преимущества трассируемости

Трассируемость между требованиями, проектом, кодом и тестовыми данными необходима для соответствия целям DO-178C. У хорошей трассируемости есть множество преимуществ.

Преимущество 1: Трассируемость необходима для прохождения аудита сертифицирующего органа. Трассируемость жизненно важна для программного проекта и для соответствия DO-178C. Без хорошей трассируемости разрушается сама концепция гарантии разработки, потому что у сертифицирующего органа нет *уверенности*. Многие цели и концепции

DO-178C и хорошей программной инженерии строятся на концепции трассируемости.

Преимущество 2: Трассируемость дает уверенность в том, что нормы соблюдены. Трассируемость важна потому, что при правильном выполнении она гарантирует, что реализованы и верифицированы все требования и что реализованы *только* требования. Это напрямую поддерживает соответствие нормам, поскольку нормы требуют доказательств того, что предусмотренная функциональность реализована.

Преимущество 3: Трассируемость имеет решающее значение для анализа влияния изменений и сопровождения. Поскольку изменения требований, проекта и кода являются неотъемлемой частью разработки программного обеспечения, инженеры должны думать о том, как сделать ПО и сопровождающие его данные ЖЦ ПО пригодными к изменению. Когда происходит изменение, трассируемость помогает определить, какие данные затронуты, какие нужно обновить и какие требуют повторной верификации.

Преимущество 4: Трассируемость помогает в управлении проектом. Актуальные данные трассировки помогают руководителям проекта понимать, что уже реализовано и верифицировано и что еще предстоит сделать.

Преимущество 5: Трассируемость помогает определить завершенность. Хорошая схема двунаправленной трассировки позволяет инженерам понимать, когда они завершили каждый элемент данных. Она также выявляет элементы данных, которые еще не были реализованы или были реализованы без движущего требования. Когда фаза (например, проектирование или разработка тестовых примеров) завершена, данные трассировки показывают, что данные ЖЦ ПО завершены, согласованы с данными предыдущей фазы и готовы к использованию в качестве входа для следующей фазы.

6.11.2 Двунаправленная трассируемость

Чтобы достичь цели реализации всех требований и только требований, необходимы два вида трассируемости: прямая трассируемость (сверху вниз) и обратная трассируемость (снизу вверх). На рисунке 6.3 показаны концепции двунаправленной трассируемости, требуемые DO-178C.

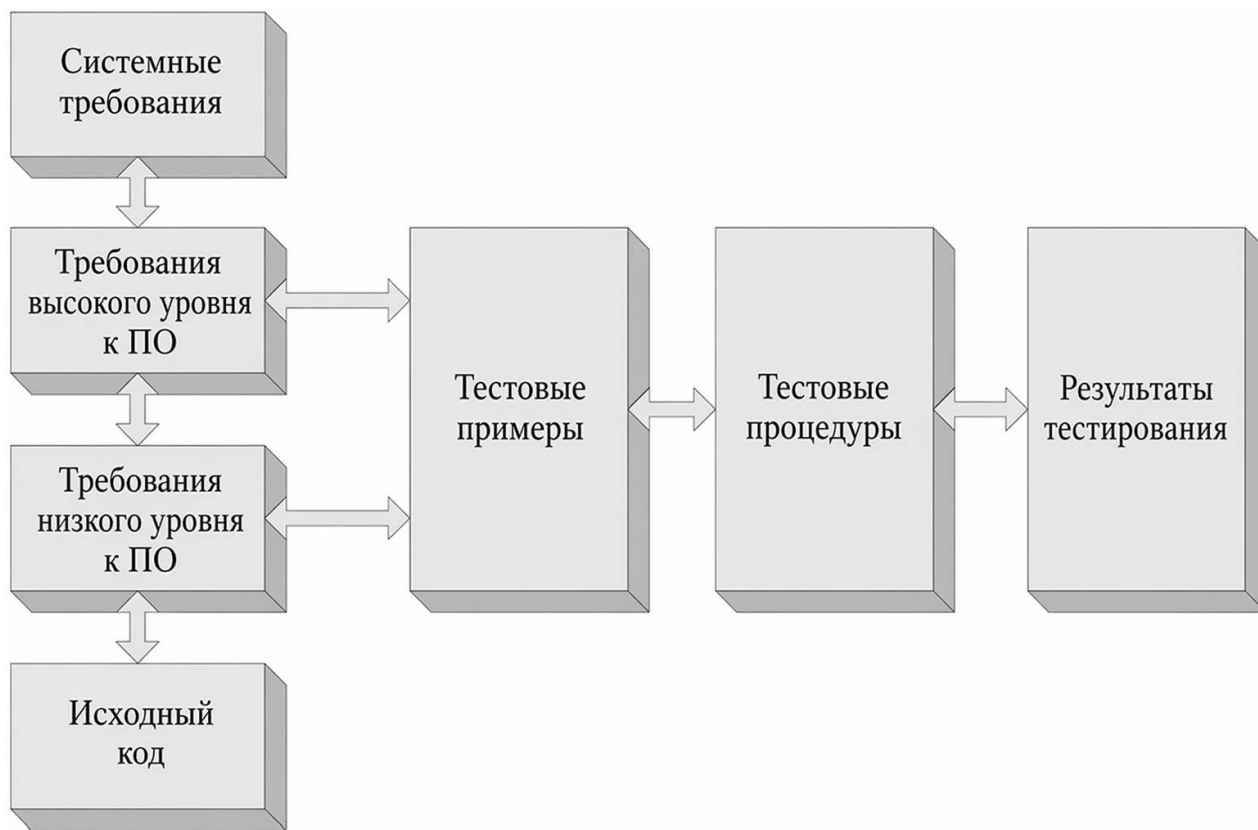


Рисунок 6.3 Двухнаправленная трассируемость между данными ЖЦ ПО.

Двухнаправленная трассируемость не возникает сама собой, ее необходимо учитывать на протяжении всего процесса разработки. Лучший способ обеспечить это – реализовывать и проверять двухнаправленную трассируемость на каждой фазе разработки, как отмечено ниже:

- Во время *рассмотрения программных требований* проверяйте трассировку сверху вниз и снизу вверх между системными требованиями и требованиями высокого уровня к ПО.
- Во время *рассмотрения описания проекта* проверяйте двухнаправленную трассировку между требованиями высокого уровня к ПО и требованиями низкого уровня к ПО.
- Во время *рассмотрения кода* проверяйте двухнаправленную трассировку между требованиями низкого уровня к ПО и исходным кодом.
- Во время *рассмотрения тестовых примеров и процедур* проверяйте двухнаправленную трассировку между тестовыми примерами и требованиями (как высокого, так и низкого уровня), а также между тестовыми примерами и тестовыми процедурами.
- Во время *рассмотрения результатов тестирования* проверяйте двухнаправленную трассировку между результатами тестирования и тестовыми процедурами.

При проведении рассмотрений следует рассматривать оба направления. То, что трассировка в одном направлении полна, еще не означает, что трассировка действительно двухнаправленна. Например, все системные требования, распределенные на программное обеспечение,

могут быть протрассированы вниз к требованиям высокого уровня к ПО; однако при этом могут существовать требования высокого уровня к ПО, не трассирующиеся вверх ни к одному системному требованию. Единственными требованиями, у которых нет родительских требований, должны быть производные требования.

Мероприятие по трассировке не должно ограничиваться только поиском полноты трассировки, но должно также оценивать техническую точность трассировки. Например, оценивая трассировку между системным требованием и его дочерними требованиями (требованиями высокого уровня к ПО), задавайте следующие вопросы:

- Полностью ли эти требования высокого уровня к ПО реализуют системное требование?
- Есть ли какая-либо часть системного требования, не отраженная в требованиях высокого уровня к ПО?
- Точна и полна ли связь между этими требованиями? Есть ли отсутствующие трассы?
- Если требования высокого уровня к ПО трассируются к нескольким системным требованиям, точна и полна ли связь между этой группой требований?
- Подходит ли уровень детализации для каждого уровня требований? Например, адекватно ли соотношение между системными требованиями и требованиями высокого уровня к ПО? Магического числа не существует, но значительное количество требований с соотношением 1:1 или 1:>10 может указывать на проблему с гранулярностью. Если имеется много трасс вида “многие-ко-многим”, уместны ли они? Избыток трасс “многие-ко-многим” (то есть дочерние требования трассируются ко многим родительским, а родительские ко многим дочерним) может указывать на проблему (это обсуждается в разделе 6.11.4).

6.11.3 DO-178C и трассируемость

DO-178B определял трассируемость как мероприятие по верификации, но был несколько расплывчат в вопросе того, как именно следует документировать эту информацию о трассировке. Большинство заявителей включают информацию о трассировке в состав своего отчета по верификации; однако некоторые включают ее в сами разработанные данные (то есть в требования, проект, код и тестовые примеры и тестовые процедуры). DO-178C по-прежнему сохраняет гибкость относительно того, где документируется информация о трассировке; однако он требует артефакт под названием данные трассировки при разработке требований, проекта, кода и тестов.

Раздел 5.5 DO-178C определяет мероприятия по трассировке, происходящие в ходе разработки программного обеспечения. Он прямо требует двунаправленной трассировки между:

1. системными требованиями, распределенными на программное обеспечение, и требо-

ваниями высокого уровня к ПО;

2. требованиями высокого уровня к ПО и требованиями низкого уровня к ПО;
3. требованиями низкого уровня к ПО и исходным кодом [3].

Таблица А-2 DO-178С определяет данные трассировки как свидетельство мероприятий по трассировке и как выход процесса разработки.

Аналогично этому, раздел 6.5 DO-178С объясняет мероприятия по трассировке в процессе верификации и требует двунаправленной трассировки между:

1. программными требованиями и тестовыми примерами;
2. тестовыми примерами и тестовыми процедурами;
3. тестовыми процедурами и результатами тестирования.

Таблица А-6 DO-178С определяет данные трассировки как выход процесса тестирования.

DO-178В не содержал термина *двунаправленная трассируемость*, хотя намекал на нее и по существу требовал ее. Раздел 6 DO-178В обсуждал прямую трассируемость (разделы 6.3.1.f, 6.3.2.f, 6.3.4.e и 6.4.4.1), тогда как цели в таблицах А-3 (цель 6), А-4 (цель 6), А-5 (цель 5) и А-7 (цели 3 и 4) DO-178В подразумевали обратную трассируемость. Таким образом, обе они требовались сертифицирующими органами. Однако DO-178С в этой области более явен. DO-178С прямо указывает на необходимость *двунаправленной трассируемости*, а также на разработку *данных трассировки*.

6.11.4 Трудности трассируемости

Как и все остальное в программной инженерии, реализация хорошей трассируемости имеет свои трудности. Ниже перечислены некоторые из распространенных трудностей.

Трудность 1: Выполнять трассировку проактивно. Большинство программистов любят решать проблемы и создавать проект или код. Однако немногие из них любят бумажную работу и ведение записей, которые сопровождают такую работу. Чтобы трассируемость была точной и полной, крайне важно документировать ее по мере разработки. Иными словами, когда пишутся требования высокого уровня к ПО, должны фиксироваться трассы к системным требованиям и от них; когда документируется проект, должны фиксироваться трассы к требованиям высокого уровня и от них; когда разрабатывается код, должна документироваться трассировка к требованиям низкого уровня и от них и так далее.

Если трассировка не выполняется проактивно автором данных, человеку, который позже не так хорошо знаком с этими данными, трудно сделать это впоследствии, потому что он может не знать ход мыслей, контекст или решения первоначальных разработчиков и будет вынужден делать предположения или допущения (которые могут оказаться неверными). Кроме того, если трассировка выполняется постфактум, в двунаправленной трассируемости обыч-

но остаются дыры. Иными словами, некоторые требования могут быть реализованы только частично, а некоторые функции могут быть реализованы, хотя в требованиях их нет. Вместо того чтобы надлежащим образом исправлять требования, инженеры, занимающиеся “уборкой” постфактум, могут выполнять частичную трассировку (чтобы все выглядело завершенным), объявлять многие требования производными (потому что у них нет родителей) или трассировать их к общим требованиям, попадая в дилемму “один-ко-слишком-многим” или “многие-ко-многим”. Особенно часто это, по-видимому, происходит тогда, когда трассировку постфактум поручают неопытным инженерам.

Трудность 2: Поддерживать данные трассировки в актуальном состоянии. Некоторые проекты очень хорошо создают первоначальные данные трассировки; однако они не обновляют их при внесении изменений. Трассируемость следует оценивать и соответствующим образом обновлять всякий раз, когда данные модифицируются. Актуальные и точные данные трассировки критичны для решений по управлению требованиями и для оценки влияния изменений (об этом говорится в главе 10).

Трудность 3: Выполнять двунаправленную трассировку. Может возникнуть соблазн считать, что все завершено, если трассируемость в одном направлении полна. Однако, как отмечалось выше, то, что полна прямая трассировка, еще не означает, что полна обратная, и наоборот. Данные трассировки должны рассматриваться как с точки зрения сверху вниз (прямой), так и с точки зрения снизу вверх (обратной).

Трудность 4: Трассы “многие-ко-многим”. Это происходит тогда, когда родительские требования трассируются к нескольким дочерним, а дочерние требования трассируются к нескольким родительским. Хотя, безусловно, существуют ситуации, когда трассировка “многие-ко-многим” точна и уместна, ее избыток обычно является симптомом проблемы. Нередко проблема заключается в том, что требования плохо организованы или что трассировка была выполнена постфактум. Хотя сертификационных рекомендаций, запрещающих трассы “многие-ко-многим”, не существует, они могут сильно запутывать разработчиков и сертифицирующий орган, а их часто очень трудно обосновывать и сопровождать. Трассировку типа “многие-ко-многим” следует минимизировать насколько это возможно.

Трудность 5: Решить, что является производным, а что нет. Производные требования могут быть трудным местом. Мне не раз приходилось оценивать проекты, где требования помечали как производные из-за отсутствия требований более высокого уровня и нежелания проекта добавлять такие требования. К производным требованиям следует относиться осторожно. Они не являются заглушками для отсутствующих требований более высокого уровня, а представляют собой детали проекта, несущественные на более высоком уровне. Производные требования не должны добавляться для компенсации отсутствующей функ-

циональности на более высоком уровне. Один из приемов, помогающих избежать неверной классификации производных требований, состоит в том, чтобы обосновывать, зачем нужно каждое требование и почему оно классифицировано как производное.[*] Если требование невозможно объяснить или обосновать, возможно, оно не нужно либо отсутствует требование более высокого уровня. Следует помнить, что все производные требования должны оцениваться командой по безопасности, а весь код должен быть трассируем к требованиям, категории “производный код” не существует.

Трудность 6: Слабые связи. Нередко встречаются требования, по которым можно спорить, являются ли они трассируемыми или производными. Они могут быть связаны с требованием более высокого уровня, но не быть его прямым результатом. Если решено включить такую трассу, полезно добавить обоснование связи между требованиями более высокого и более низкого уровня. За неимением лучшего термина я называю такие спорные трассы слабыми связями. Краткая заметка о том, почему эти слабые связи включены, помогает всем пользователям требований лучше понять связь и оценивать влияние будущих изменений. Такие заметки также помогают сертификации и сопровождению программного обеспечения, поясняя связи, которые могут быть неочевидны. Объяснение не должно быть многословным; обычно вполне достаточно одного-двух коротких предложений. Даже если требование классифицировано как производное, все равно рекомендуется упомянуть эту связь, поскольку позднее это может помочь анализу влияния изменений и управлению изменениями (то есть поддерживает модифицируемость).

Трудность 7: Неявная трассируемость. Некоторые проекты используют концепцию неявной трассируемости. Например, трассировка может предполагаться по соглашению об именовании или по компоновке документа. Такие подходы принимались сертифицирующими органами. Новая трудность состоит в требовании DO-178C по наличию данных трассировки. Неявная трассировка встроена в сами данные и не приводит к появлению отдельного артефакта трассировки. Поэтому с неявной трассировкой следует обращаться осторожно. Вот несколько рекомендаций:

- Включайте правила трассировки в стандарты и планы, чтобы разработчики знали, чего от них ожидают.
- Хорошо документируйте рассмотрения трассировки, чтобы обеспечить точность и полноту неявной трассировки.
- Указывайте этот подход в программных планах и получайте его одобрение со стороны сертифицирующего органа.
- Будьте последовательны. Не смешивайте неявную и явную трассировку в одном разделе (если это не задокументировано должным образом).

Трудность 8: Трассы тестов на робастность. Иногда разработчики или тестировщики утверждают, что тестирование на робастность не нужно трассировать к требованиям, поскольку они пытаются *сломать* программное обеспечение, а не доказать предусмотренную функциональность. Они утверждают, что если ограничить тестировщиков только требованиями, то можно пропустить какую-то слабость ПО. Как будет обсуждаться в главе 9, установка на то, чтобы *сломать систему*, важна для эффективного тестирования ПО. Тестировщики не должны ограничиваться только требованиями, когда продумывают свою стратегию тестирования. Однако тестировщики должны выявлять неполноту требований, а не создавать тесты на робастность, которые не трассируются к требованиям. Очень часто тестировщики выявляют отсутствующие сценарии, которые должны быть отражены в требованиях. Настоятельно рекомендуется, чтобы тестировщики участвовали в рассмотрении требований и проекта, заранее выявляя потенциальные слабости требований. Аналогично этому, разработчикам может быть полезно участвовать в рассмотрении тестов, чтобы быстро закрывать пробелы в требованиях и гарантировать, что тестировщики правильно понимают требования.

Литература

1. N. Leveson, *Safeware: System Safety and Computers* (Reading, MA: Addison-Wesley, 1995).
2. IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std-6101990 (Los Alamitos, CA: IEEE Computer Society Press, 1990).
3. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
4. K. E. Wiegers, *Software Requirements*, 2nd edn. (Redmond, WA: Microsoft Press, 2003).
5. D. Leffingwell, Calculating the return on investment from more effective requirements, *American Programmer* 10(4), 13-16, 1997.
6. Standish Group Study (1995), cited in I. F. Alexander and R. Stevens, *Writing Better Requirements* (Harlow, U.K.: Addison-Wesley, 2002).
7. D. L. Lempia and S. P. Miller, *Requirements Engineering Management Findings Report*, DOT/FAA/AR-08/34 (Washington, DC: Office of Aviation Research, June 2009).
8. SearchSoftwareQuality.com, Software quality resources. <http://searchsoftwarequality.techtarget.com/case> (дата обращения: 01.05.2012).
9. K. E. Wiegers, *More about Software Requirements* (Redmond, WA: Microsoft Press, 2006).
10. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Washington, DC:

RTCA, Inc., December 2011).

11. Certification Authorities Software Team (CAST), Merging high-level and lowlevel requirements, Position Paper CAST-15 (February 2003).
12. A. M. Davis, *Software Requirements* (Upper Saddle River, NJ: PrenticeHall, 1993).
13. D. L. Lempia and S. P. Miller, *Requirements Engineering Management Handbook*, DOT/FAA/AR-08/32 (Washington, DC: Office of Aviation Research, June 2009).

Рекомендуемое чтение

1. K. E. Wiegers, *Software Requirements*, 2nd edn. (Redmond, WA: Microsoft Press, 2003).
Хотя эта книга не написана специально для программного обеспечения, критичного по безопасности, она является превосходным ресурсом для авторов требований.
 2. D. L. Lempia and S. P. Miller, *Requirements Engineering Management Handbook*, DOT/FAA/AR-08/32 (Washington, DC: Office of Aviation Research, June 2009). Это руководство было подготовлено при поддержке FAA и содержит ценные рекомендации для тех, кто пишет требования к программному обеспечению, критичному по безопасности.
- *У инженера по требованиям вероятно будут и другие обязанности, но эта глава сосредоточена на его или ее роли как автора требований; поэтому термин *инженер по требованиям* используется на протяжении всей главы.
- *Вариант использования (use case) – это методология, применяемая при анализе для выявления, уточнения и организации требований. Каждый вариант использования состоит из набора возможных последовательностей взаимодействий между программным обеспечением и его пользователями в определенной среде и связан с определенной целью. “О варианте использования можно думать как о наборе возможных сценариев, связанных с определенной целью; действительно, вариант использования и цель иногда считаются синонимами” [8]. Варианты использования могут применяться для организации функциональных требований, моделирования пользовательских взаимодействий, фиксации сценариев от событий к целям, описания основного потока событий и описания нескольких уровней функциональности [8].
- *Скобки добавлены для ясности.
- *Группа органов сертификации по программному обеспечению (Certification Authorities Software Team, CAST) – это группа международных органов по сертификации, стремящихся гармонизировать свои позиции по бортовому ПО и бортовой электронной аппаратуре в документах CAST.

*Если не указано иное, эти положения основаны на разделах 6.3.1 и 11.9 DO-178C.

† Согласно разделу 8.1.c DO-178C.

*Обычно персонал по качеству и сертификации участвует по своему усмотрению. Кроме того, специалисты по безопасности могут рассматривать требования, не участвуя в заседании по рассмотрению. Обязательные рецензенты должны быть четко указаны в планах.

*Раздел 5.1.2.h DO-178C определяет это как мероприятие.

Глава 7. Проектирование программного обеспечения

Сокращения

Сокращение	Расшифровка
CSPEC	Control specification (спецификация управления)
TBY	High-level requirement (требование высокого уровня к ПО)
THY	Low-level requirement (требование низкого уровня к ПО)
PSPEC	Process specification (спецификация процесса)
UML	Unified Modeling Language (унифицированный язык моделирования)

7.1 Обзор проектирования программного обеспечения

В области проектирования DO-178C (KT-178C) несколько отходит от остальной литературы по разработке программного обеспечения. В DO-178B, а затем и в DO-178C, поясняется, что проект ПО включает архитектуру ПО и требования низкого уровня к ПО (low-level requirements, THY). Архитектура ПО как часть проектирования обычно понятна без дополнительных пояснений; значительную путаницу вызвал именно термин «требования низкого уровня». В ходе обсуждений в комитете DO-178C предпринималась попытка скорректировать терминологию так, чтобы она лучше соответствовала другим областям и общепринятым методам программной инженерии, но согласия по этому изменению достигнуто не было. Поэтому этот термин сохранился и рассматривается в данной главе.

Проект служит чертежом для этапа реализации программного обеспечения. Он описывает как то, *как* будет устроено ПО, то есть его архитектуру, так и то, *как* оно будет выполнять требуемую функциональность, то есть через требования низкого уровня к ПО. Чтобы понять указания DO-178C по проектированию, важно различать два элемента проекта ПО: архитектуру и требования низкого уровня к ПО.

7.1.1 Архитектура программного обеспечения

DO-178C определяет *архитектуру программного обеспечения* так: «Структура ПО, выбранная для реализации требований к ПО» [1]. Роджер Прессман дает более полное определение:

В простейшей форме архитектура представляет собой структуру или организацию программных компонентов (модулей), способ взаимодействия этих компонентов и структуру данных, используемых компонентами. В более широком смысле компоненты, однако, можно обобщить так, чтобы они представляли основные элементы системы и их взаимодействия [2].

Архитектура является важнейшей частью процесса проектирования. Ниже приведены неко-

торые моменты, которые следует учитывать при документировании архитектуры:

Во-первых, соответствие DO-178C требует, чтобы архитектура была совместима с требованиями. Следовательно, необходим некоторый способ обеспечить такую совместимость. Часто для этого обеспечивают трассируемость или создают отображение между требованиями и архитектурой.

Во-вторых, архитектура должна документироваться в ясном и согласованном формате. Важно учитывать и программиста, который будет использовать проект для реализации кода, и разработчиков, которые в будущем будут сопровождать программное обеспечение и его проект. Архитектура должна быть определена ясно, чтобы ее можно было точно реализовать и сопровождать.

В-третьих, архитектура должна быть задокументирована таким образом, чтобы ее можно было при необходимости обновлять и, возможно, реализовывать итерационно. Это может быть нужно для поддержки итеративной или эволюционной разработки, вариантов конфигурации или подхода к обеспечению безопасности.

В-четвертых, существуют различные архитектурные стили. Для большинства стилей архитектура включает компоненты и соединители. Однако тип этих компонентов и соединителей зависит от используемого архитектурного подхода. Большая часть авиационного программного обеспечения реального времени использует функциональную структуру. В этом случае компоненты представляют функции, а соединители показывают интерфейсы между функциями (либо в форме данных, либо в форме управления).

7.1.2 Требования низкого уровня к ПО

DO-178C определяет требования низкого уровня к ПО так: «Требования к программному обеспечению, разработанные на основе требований высокого уровня, производных требований и проектных ограничений, по которым исходный код может быть непосредственно реализован без дополнительной информации» [1]. Требования низкого уровня к ПО представляют собой декомпозицию требований высокого уровня к ПО (high-level requirements, ТВУ) до такого уровня, с которого код уже можно писать напрямую. По сути, DO-178C переносит детальную инженерную проработку на этап проектирования, а не кодирования. Если проект задокументирован надлежащим образом, работа по кодированию должна быть сравнительно прямолинейной. Эта философия не лишена спорности. Необходимость двух уровней требований к ПО, высокого и низкого уровня, активно обсуждалась. Ниже приведены 10 положений, которые следует иметь в виду при планировании и написании требований низкого уровня к ПО:

Положение 1: Требования низкого уровня являются деталями проекта. Слово требования может вводить в заблуждение, поскольку требования низкого уровня являются частью

проектирования. Полезно думать о них как о шагах реализации, которым должен следовать программист. Иногда такие требования даже представляют в виде псевдокода или моделей. Некоторые проекты требуют использовать в формулировках слово *должен (shall)*, а некоторые нет. Мне доводилось видеть, как оба подхода успешно применялись на практике.

Положение 2: Требования низкого уровня должны иметь уникальную идентификацию. Поскольку требования низкого уровня должны трассироваться вверх к требованиям высокого уровня и вниз к коду, они должны быть уникально идентифицированы. Именно поэтому некоторые организации предпочитают включать слово *должен (shall)* в текст требований.

Положение 3: Требования низкого уровня должны обладать атрибутами качества, описанными в главах 2 и 6. Однако требования низкого уровня сосредоточены на *том, как*, а не на *том, что*. Это детали реализации, и они должны углубляться в подробности того, как именно программное обеспечение будет реализовано для выполнения функциональности, задокументированной в требованиях высокого уровня.

Положение 4: Требования низкого уровня должны быть верифицируемыми. Одна из причин, по которым слово *требования* было сохранено в DO-178C, состоит в том, что для уровней А, В и С требования низкого уровня должны тестироваться. Это важно учитывать при их написании.

Положение 5: Иногда требования низкого уровня могут не требоваться вообще. В некоторых проектах требования высокого уровня бывают настолько подробными, что код можно писать непосредственно по ним. Это исключение, а не правило. И, как отмечалось в главе 6, такой подход не рекомендуется. Требования высокого уровня должны быть сосредоточены на функциональности, тогда как требования низкого уровня сосредоточены на реализации. Однако если требования высокого уровня действительно достаточно подробны, а это означает, что в них, вероятно, уже смешано *что* и *как*, раздел 5.0 DO-178C все же допускает один уровень требований к программному обеспечению, если этот единственный уровень способен удовлетворить как целям требований высокого уровня, так и целям требований низкого уровня, то есть целям 1, 2, 4 и 5 таблицы А-2 DO-178C, таблицы А-3 и таблицы А-4. Кроме того, содержание этого единственного уровня требований в сочетании с архитектурой ПО должно удовлетворять указаниям, приведенным в разделах 11.9, посвященном требованиям к ПО, и 11.10, посвященном проекту ПО, стандарта DO-178C. Однако следует иметь в виду, что сертифицирующие органы принимают такой подход не всегда, и при недостаточной координации он может привести к перезапуску проекта. Если для вашего проекта этот подход выглядит реализуемым, обязательно опишите его в планах, обоснуйте, почему он сработает, и подробно покажите, как будут удовлетворены цели DO-178C.

Положение 6: Иногда требования низкого уровня могут не требоваться в некоторых об-

лостях. В отдельных проектах требования высокого уровня в большинстве областей могут быть достаточно подробными, чтобы по ним сразу писать код, но в некоторых областях может понадобиться дополнительная детализация. Иными словами, часть требований высокого уровня нужно декомпозировать в требования низкого уровня, а часть нет. Если используется такой подход, должно быть ясно, какие требования относятся к высокому уровню, а какие к низкому. Также должно быть ясно, когда требования высокого уровня не будут дальше декомпозироваться, чтобы программист понимал, какие требования служат основой для работ по кодированию. На практике этот подход может оказаться весьма коварным, поэтому применять его следует с крайней осторожностью.

Положение 7: Чем выше критичность программного обеспечения, тем более подробные требования низкого уровня обычно требуются. По моему опыту, чем выше критичность ПО, особенно для уровней А и В, тем более подробными должны быть требования низкого уровня, чтобы полностью описать требования и получить необходимое структурное покрытие. Чем критичнее ПО, тем строже критерии структурного покрытия. Структурное покрытие рассматривается в главе 9; однако это стоит учитывать уже на этапе проектирования.

Положение 8: С производными требованиями низкого уровня необходимо обращаться осторожно. В ходе этапа проектирования могут быть выявлены некоторые производные требования низкого уровня. DO-178С определяет производные требования так: «Требования, создаваемые процессами разработки ПО, которые (а) не трассируются непосредственно на требования более высокого уровня и/или (б) задают поведение сверх того, которое определено системными требованиями или требованиями высокого уровня к ПО» [1]. Производные требования низкого уровня не предназначены для того, чтобы латать дыры в требованиях высокого уровня; вместо этого они представляют детали реализации, которые еще не были известны на этапе работы с требованиями. Производные требования низкого уровня не трассируются вверх к требованиям высокого уровня, но будут трассироваться вниз к коду. Каждое такое производное требование должно быть задокументировано, помечено как производное, обосновано с точки зрения необходимости и оценено командой по анализу безопасности, чтобы убедиться, что оно не нарушает никаких системных допущений или допущений безопасности. Чтобы поддержать команду по безопасности, обоснование должно быть написано так, чтобы человек, не знакомый с деталями проекта, мог понять требование и оценить его влияние на безопасность и общую функциональность системы.

Положение 9: Требования низкого уровня обычно имеют текстовую форму, но могут представляться и как модели. Требования низкого уровня часто выражаются в текстовом виде с таблицами и графикой, когда это нужно для передачи деталей. Как будет обсуждаться в

главе 14, они могут быть представлены в виде моделей. В этом случае модель будет классифицироваться как *модель проекта*, и к ней будут применяться указания DO-331.

Положение 10: Требования низкого уровня могут представляться в виде псевдокода. Иногда требования низкого уровня представляют в виде псевдокода или дополняют псевдокодом. В DO-248C, в ответе на часто задаваемый вопрос No. 82 (FAQ No. 82) под названием «Если псевдокод используется как часть требований низкого уровня, какие вопросы необходимо учитывать?» рассматриваются сертификационные вопросы, связанные с использованием псевдокода в качестве требований низкого уровня [3]. Когда требования низкого уровня представлены в виде псевдокода, возникают следующие опасения [3]:

- Такой подход может привести к слишком большому скачку детализации между требованиями высокого уровня и требованиями низкого уровня, из-за чего будет трудно выявлять непредусмотренную и отсутствующую функциональность.
- Такой подход может привести к недостаточной детализации архитектуры, что повлияет на верификацию, включая анализ межкомпонентной связности по данным и по управлению.
- Может быть трудно обеспечить уникальную идентификацию требований низкого уровня.
- Двухнаправленная трассируемость к требованиям высокого уровня и от них может оказаться затруднительной. Выполнение структурного покрытия с помощью тестирования на низком уровне, как правило, недостаточно, поскольку код и псевдокод слишком похожи; такой процесс тестирования не позволяет эффективно выявлять ошибки, обнаруживать отсутствующую функциональность или находить непредусмотренную функциональность.

7.1.3 Компоновка проектной документации

Компоновка проектной документации различается от проекта к проекту. В одних проектах архитектура и требования низкого уровня к ПО объединяются, в других помещаются в полностью отдельные документы. В некоторых проектах они находятся в одном документе, но в разных разделах. DO-178C не диктует предпочтительный вариант компоновки. Часто он зависит от используемой методологии. Независимо от принятого решения о компоновке, связь между архитектурой и требованиями должна быть ясной. Проект, включая требования низкого уровня к ПО, будет использоваться теми, кто реализует программное обеспечение; следовательно, при документировании проекта необходимо помнить о программистах.

7.2 Подходы к проектированию

Существует множество методов, которые проектировщики используют для моделирования архитектуры и поведения программного обеспечения. В авиационном ПО применяются два подхода к проектированию: структурный и объектно-ориентированный. В некоторых проектах сочетаются понятия обоих подходов.

7.2.1 Структурное проектирование (традиционный подход)

Структурное проектирование широко применяется для встроенного программного обеспечения реального времени и использует некоторые или все из следующих представлений:

Примечание переводчика. Ниже перечислены не обязательные документы, а возможные виды одного и того же проекта. Если представить обычную прошивку на С для микроконтроллера, то эти представления отвечают на разные практические вопросы: что приходит на входы, что уходит на выходы, какие функции или задачи это обрабатывают, где данные временно лежат, какие условия выбирают ветку алгоритма и какие состояния есть у программы. Это не замена коду и не пересказ файлов .с; это способ заранее описать устройство программы так, чтобы потом было понятно, почему код должен быть именно таким.

- *Диаграмма контекста данных* – диаграмма верхнего уровня, которая описывает функциональное поведение программного обеспечения и показывает входные и выходные данные ПО.

Примечание переводчика. Для прошивки на С это вид «снаружи внутрь». Внутри пока не видно ни `main()`, ни драйверов, ни задач RTOS. Видно только, что, например, программа читает АЦП, принимает кадр по CAN, получает дискретный вход, считывает параметр конфигурации, а наружу выдает ШИМ, команду по UART, статусный бит или сообщение диагностики. Такая диаграмма фиксирует границу ПО: что считается входом программы и что считается ее выходом.

- *Диаграмма потоков данных* – графическое представление процессов, выполняемых программным обеспечением, показывающее поток данных между процессами. Она является декомпозицией диаграммы контекста данных. Диаграмма потоков данных обычно представляется на нескольких уровнях, причем каждый следующий уровень дает больше подробностей. Диаграммы потоков данных включают процессы, потоки данных и хранилища данных.

Примечание переводчика. Слово «процесс» здесь не обязательно означает процесс операционной системы. В маленькой программе на С без ОС таким процессом может быть просто логический шаг обработки: `read_inputs()`, `filter_sensor_value()`,

`compute_command()`, `update_outputs()`. «Хранилище данных» тоже не обязательно база данных. Для микроконтроллера это может быть глобальная структура состояния, кольцевой буфер UART, очередь сообщений RTOS, массив последних измерений, калибровочная таблица во flash или набор переменных, которые один модуль записывает, а другой читает. Диаграмма показывает не порядок вызовов, а путь данных: откуда значение взялось, через какую обработку прошло и где оказалось.

- *Спецификация процесса (PSPEC)* – сопровождает диаграмму потоков данных и показывает, как по заданным входным данным формируются выходы процессов [4].

Примечание переводчика. PSPEC – это место, где блок с диаграммы перестает быть просто прямоугольником. Например, если на диаграмме есть процесс «вычислить команду двигателя», то в PSPEC можно описать: взять отфильтрованное значение датчика, ограничить его диапазоном, применить коэффициент из калибровки, проверить признак отказа и сформировать команду. В коде это может стать одной функцией, несколькими статическими функциями в модуле или частью периодической задачи. Важно, что PSPEC описывает преобразование данных, а не расписание вызовов функций.

- *Диаграмма контекста управления* – диаграмма верхнего уровня, которая показывает управление системой путем задания управляющих интерфейсов между системой и ее окружением.
- *Диаграмма потока управления* – та же диаграмма, что и диаграмма потоков данных, но в ней определяется не поток данных, а поток управления через систему.

Примечание переводчика. Это место легко спутать с деревом вызовов, но речь о другом. Дерево вызовов для C показывает, что `main()` вызывает `init()`, затем `while(1)` вызывает `control_step()`, а `control_step()` вызывает `compute_command()`. Диаграмма потока управления показывает, почему выбирается та или иная обработка: пришло прерывание от таймера, установлен флаг `sensor_valid`, состояние автомата равно `ARMED`, получена команда `START`, истек тайм-аут, сработала защита. В коде это обычно превращается в `if`, `switch`, проверки флагов, обработчики прерываний, таблицы переходов состояний и события RTOS. Поэтому это не «кто кого вызывает», а «какое событие или условие разрешает, запрещает или переключает выполнение».

- *Спецификация управления (CSPEC)* – сопровождает диаграмму потока управления и показывает, как по заданным входным данным формируются выходы процессов [4].

Примечание переводчика. CSPEC можно читать как описание «диспетчера решений». Например: если система в состоянии `IDLE` и пришла команда `START`, перейти в `RUN`; если в `RUN` пропал валидный датчик, перейти в `FAULT`; если в `FAULT` пришел сброс и ди-

агностика успешна, вернуться в IDLE. В С это часто выглядит как конечный автомат на switch (state), таблица переходов или набор проверок флагов. PSPEC отвечает за вычисления вроде «как посчитать выходное значение», а CSPEC – за то, «когда эту обработку запускать, какой режим выбрать и в какое состояние перейти».

- *Таблица решений (decision table, также truth table)* – показывает комбинации решений, принимаемых на основе заданных входных данных.

Примечание переводчика. В приземленном виде это таблица для сложного if. Например, есть три условия: датчик валиден, команда оператора разрешена, питание в норме. Таблица перечисляет комбинации этих условий и показывает, что делать в каждой комбинации: включить выход, заблокировать выход, выдать отказ, остаться в текущем состоянии. В С такая таблица может быть реализована обычными условиями, switch, битовой маской признаков или настоящей таблицей правил. Ее смысл – явно показать все комбинации, а не надеяться, что читатель восстановит их из вложенных if на три страницы.

- *Диаграмма переходов состояний* – иллюстрирует поведение системы, показывая ее состояния и события, которые вызывают переход системы из одного состояния в другое. В некоторых проектах это может быть представлено таблицей переходов состояний вместо диаграммы.
- *Спецификация времени отклика* – иллюстрирует внешние времена отклика, которые необходимо задать. Она может включать событийно-управляемые, непрерывные или периодические времена отклика. В ней определяются входное событие, выходное событие и время отклика для каждого внешнего входного сигнала [4].
- *Блок-схема* – графически представляет последовательность действий и решений программного обеспечения.
- *Структурная схема* – иллюстрирует разбиение системы на модули, показывая их иерархию, организацию и взаимодействие [5].
- *Дерево вызовов (call tree, также call graph)* – иллюстрирует отношения вызова между программными модулями, функциями или процедурами.
- *Словарь данных* – определяет данные и управляющую информацию, которые проходят через систему. Обычно для каждого элемента данных включает следующую информацию: имя, описание, частоту поступления, диапазон, разрешение, единицы измерения, где и как используется и т.д.

Примечание переводчика. Для C это полезно читать как паспорт данных. Например: `adc_raw` - безразмерный код АЦП 0..4095, обновляется раз в 1 мс; `temperature_c` - температура в градусах Цельсия, диапазон от -40 до 125, получается после пересчета `adc_raw`; `motor_cmd` - команда исполнительному механизму в процентах, диапазон 0..100, передается в `pwm_set()`. В словарь попадают и управляющие признаки: `sensor_valid`, `fault_latched`, `mode`. Это помогает не путать сырые и пересчитанные значения, единицы измерения, диапазоны, частоты обновления и владельца данных.

- *Текстовые детали* – описывают детали реализации, например требования низкого уровня к ПО.
- *Диаграмма задач* – показывает характеристики задач, например являются ли они спорадическими или периодическими, процедуры задач, входы и выходы каждой задачи, а также любые взаимодействия с операционной системой, такие как семафоры, сообщения и очереди.

Примечание переводчика. Если используется RTOS, задача – это обычно конкретная функция-тело задачи, например `ControlTask(void *arg)`, `CommsTask(void *arg)` или `DiagnosticsTask(void *arg)`. Диаграмма задач показывает, что `ControlTask` запускается каждые 10 мс, читает последние измерения, пишет команду на выход; `CommsTask` просыпается по сообщению из очереди CAN; обработчик прерывания только кладет байты в кольцевой буфер и будит задачу. Если RTOS нет, близкую роль может играть главный цикл с периодическими шагами и обработчиками прерываний. Смысл диаграммы – показать не алгоритм вычисления, а кто, когда и с кем взаимодействует во время выполнения.

7.2.2 Объектно-ориентированное проектирование

Методы объектно-ориентированного проектирования могут использовать следующие представления [2]:

- *Вариант использования (use case)* – сопровождает требования и графически и текстуально поясняет, как пользователь взаимодействует с системой при определенных обстоятельствах. Он определяет акторов, то есть людей или устройства, использующие систему, и описывает, как актер взаимодействует с системой.
- *Диаграмма активности (activity diagram)* – дополняет вариант использования графическим представлением потока взаимодействия внутри сценария. Она показывает поток управления между действиями, которые выполняет система. Диаграмма активности похожа на блок-схему, но, в отличие от нее, также показывает параллельные потоки.
- *Диаграмма дорожек (swimlane diagram)* – разновидность диаграммы активности; по-

казывает поток действий, описанных в варианте использования, и одновременно указывает, какой актер отвечает за действие, описываемое соответствующей активностью. По сути, она показывает действия каждого актора параллельно.

- *Диаграмма состояний (state diagram)* – подобно описанной ранее диаграмме переходов состояний, объектно-ориентированная диаграмма состояний показывает состояния системы, действия, выполняемые в зависимости от этих состояний, и события, приводящие к смене состояния.
- *Иерархическая диаграмма состояний (state chart)* – расширение диаграммы состояний с добавлением информации об иерархии и параллелизме.
- *Диаграмма классов (class diagram)* – подход унифицированного языка моделирования (Unified Modeling Language, UML), моделирующий классы, включая их атрибуты, операции, отношения и ассоциации с другими классами, путем предоставления статического, то есть структурного, представления системы.
- *Диаграмма последовательностей (sequence diagram)* – показывает взаимодействия между объектами во время выполнения задачи, включая временной порядок, в котором между объектами передаются сообщения для выполнения этой задачи.
- *Модель отношений объектов (object-relationship model)* – графическое представление связей между классами.
- *Модель «класс-ответственность-соисполнитель» (class-responsibility-collaborator)* – предоставляет способ выявить и организовать классы, относящиеся к требованиям. Каждый класс представляется в виде блока, который иногда называют *карточкой*; каждый такой блок включает имя класса, обязанности класса и соисполнителей. Обязанности – это атрибуты и операции, относящиеся к данному классу. Соисполнители – это те классы, которые должны предоставить информацию другому классу, чтобы он мог выполнить свою обязанность. Взаимодействие представляет собой либо запрос информации, либо запрос на выполнение некоторого действия.

В главе 15 приводится дополнительная информация об объектно-ориентированной технологии.

7.3 Характеристики хорошего проекта

DO-178С предоставляет гибкость в документировании проекта. Вместо того чтобы подробно разбирать методы проектирования, которые и без того описаны во множестве других книг, рассмотрим характеристики, которыми должен обладать хороший проект программного обеспечения.

Характеристика 1: Абстракция. Хороший проект реализует концепцию абстракции на

нескольких уровнях. Абстракция – это процесс определения программы или данных через представление, близкое к их смыслу, при одновременном сокрытии деталей реализации. Абстракция стремится сократить число деталей и вынести их за скобки, чтобы проектировщик мог сосредоточиться на нескольких понятиях одновременно. Когда абстракция применяется на каждом иерархическом уровне разработки, это позволяет каждому уровню работать только с теми деталями, которые существенны именно для него. Желательны как процедурная, так и абстракция данных.

Характеристика 2: Модульность. Модульный проект – это проект, в котором программное обеспечение логически разделено на элементы, модули или подсистемы, часто называемые *компонентами*. (*Компонент* может быть отдельным программным модулем или группой связанных программных модулей.) Система в целом делится по возможностям или функциям. Каждый *компонент* отвечает за конкретную возможность или функцию. Когда возможности и функциональность разделены на более мелкие управляемые компоненты, общую задачу становится проще решать. Прессман пишет:

Проект, а затем и получающуюся по нему программу, разбивают на модули, чтобы разработку было проще планировать; чтобы можно было заранее выделять части ПО и поставлять их поэтапно; чтобы изменения было проще вносить; чтобы тестирование и отладка выполнялись эффективнее; и чтобы долговременное сопровождение обходилось без серьезных побочных эффектов [2].

Когда проект корректно разбит на модули, сравнительно просто понять назначение каждого компонента, проверить корректность каждого компонента, разобраться во взаимодействии компонентов и оценить влияние каждого компонента на структуру и работу программного обеспечения в целом [6].

Характеристика 3: Сильная внутренняя связность компонента (strong cohesion). Чтобы система была по-настоящему модульной, проектировщик стремится к функциональной независимости каждого компонента. Для этого важно различать внутреннюю связность компонента (*cohesion*) и межкомпонентную связность (*coupling*). Хороший проект стремится к сильной внутренней связности компонентов и слабой межкомпонентной связности. «Связность можно рассматривать как клей, который удерживает компонент как единое целое» [7]. Хотя DO-178C не требует оценивать внутреннюю связность компонента, ее следует учитывать при проектировании, потому что она влияет на общее качество проекта. Внутренняя связность компонента показывает его цельность и действует как цепь, удерживающая вместе выполняемые им действия [5]. Йордон и Константин выделяют семь уровней внутренней связности компонента; по мере движения вниз по списку внутренняя связность ослабевает [8]:

- **Функциональная внутренняя связность (*functional cohesion*):** Все элементы работают на одну функцию; каждый элемент участвует только в выполнении одной задачи.
- **Последовательная внутренняя связность (*sequential cohesion*):** Компонент состоит из последовательности элементов, где выход одного элемента служит входом для следующего.
- **Коммуникационная внутренняя связность (*communicational cohesion*):** Элементы компонента используют одни и те же входные или выходные данные, но порядок их выполнения несущественен.
- **Процедурная внутренняя связность (*procedural cohesion*):** Элементы компонента участвуют в различных и, возможно, не связанных между собой действиях, которые должны выполняться в заданном порядке.
- **Временная внутренняя связность (*temporal cohesion*):** Элементы компонента функционально независимы, но их действия связаны по времени, то есть выполняются одновременно.
- **Логическая внутренняя связность (*logical cohesion*):** Элементы включают задачи, логически связанные между собой. Логически связный компонент содержит несколько действий одного общего вида, а пользователь выбирает, что именно требуется.
- **Случайная внутренняя связность (*coincidental cohesion*):** Элементы объединены в компонент случайным образом. Между ними нет содержательной связи.

Характеристика 4: Слабая межкомпонентная связность (*loose coupling*). Межкомпонентная связность (*coupling*) – это степень взаимозависимости между двумя компонентами. Хороший проект минимизирует межкомпонентную связность, устраняя ненужные связи, уменьшая число необходимых связей и ослабляя тесноту необходимых связей [5]. Слабая межкомпонентная связность помогает уменьшить волновой эффект при изменении компонента, поскольку такой компонент легче понять и адаптировать. Поэтому для эффективного и модульного проекта желательна слабая межкомпонентная связность.

Примечание переводчика. В общей программной инженерии *cohesion* и *coupling* – разные понятия. *Cohesion* описывает, насколько цельно устроен сам компонент; здесь это называется внутренней связностью компонента. *Coupling* описывает зависимости между компонентами; здесь это называется межкомпонентной связностью. В русскоязычной литературе *coupling* часто переводят как «сцепление», но в контексте DO-178C/КТ-178 закреплены выражения «связность по данным» и «связность по управлению». Поэтому в этом переводе используется единый термин «связность» с обязательным уточнением: внутренняя связность компонента или межкомпонентная связность.

При первом употреблении или там, где возможна неоднозначность, английский оригинал оставляется в скобках.

DO-178C определяет два вида межкомпонентной связности (*coupling*) [1]:

- *Межкомпонентная связность по данным (data coupling)*: «Зависимость компонента ПО от данных, не находящихся под исключительным контролем данного компонента».
- *Межкомпонентная связность по управлению (control coupling)*: «Способ или степень влияния, оказываемого одним компонентом программы на работу другого компонента».

Однако в литературе по программной инженерии выделяют шесть типов межкомпонентной связности, перечисленных ниже от наиболее сильной к наиболее слабой [5,7]:

- *Межкомпонентная связность по содержимому (content coupling)*: Один компонент использует или изменяет внутренние данные другого компонента.
- *Общая межкомпонентная связность (common coupling)*: Два компонента обращаются к одной и той же области глобальных данных. Иными словами, компоненты совместно используют ресурсы.
- *Внешняя межкомпонентная связность (external coupling)*: Компоненты взаимодействуют через внешнюю среду, например файл или базу данных.
- *Межкомпонентная связность по управлению (control coupling, не в смысле определения DO-178C)*: Один компонент управляет выполнением другого компонента, передавая ему необходимую управляющую информацию.
- *Межкомпонентная связность по структуре данных (stamp coupling)*: Два компонента обращаются к одной и той же структуре данных. Иногда это называют *межкомпонентной связностью по структуре данных*.
- *Межкомпонентная связность по данным (data coupling, не в смысле определения DO-178C)*: Два компонента взаимодействуют, передавая элементарные параметры, например однородную таблицу или отдельное поле.

К сожалению, DO-178C использует выражения *data coupling* и *control coupling* иначе, чем многие учебники по программной инженерии. Употребление термина «межкомпонентная связность по данным» в DO-178C сопоставимо с общепринятыми в программной инженерии понятиями межкомпонентной связности по данным, по структуре данных, общей и внешней межкомпонентной связности. Употребление термина «межкомпонентная связность по управлению» в DO-178C охватывается понятиями межкомпонентной связности по управлению и по содержимому [9]. Анализ межкомпонентной связности по данным и по управлению рассматривается в главе 9, поскольку эти анализы относятся к этапу вери-

фикации.

Хорошо спроектированный программный продукт стремится к слабой межкомпонентной связности и сильной внутренней связности компонентов. Реализация этих характеристик помогает упростить взаимодействие между программистами, облегчить доказательство корректности компонентов, уменьшить распространение последствий изменений между компонентами при изменении одного из них, сделать компоненты более понятными и сократить число ошибок [7].

Характеристика 5: Соккрытие информации. Соккрытие информации тесно связано с понятиями абстракции, внутренней связности компонента и межкомпонентной связности. Оно способствует модульности и повторному использованию. Концепция сокращения информации предполагает, что «модули [компоненты] должны быть специфицированы и спроектированы так, чтобы информация (например, алгоритмы и данные), содержащаяся внутри модуля [компонента], была недоступна другим модулям [компонентам], которым такая информация не требуется» [2].[*]

Характеристика 6: Сниженная сложность. Хороший проект стремится уменьшать сложность: большая система разбивается на меньшие, четко определенные подсистемы или функции. Это связано с абстракцией, модульностью и сокращением информации, но само по себе требует от проектировщика сознательных усилий. Проект должен быть документирован прямо и понятно. Если сложность слишком велика, проектировщику следует искать другие способы распределить обязанности функции между несколькими более мелкими функциями. Чрезмерно сложные проекты приводят к ошибкам и серьезным последствиям при внесении изменений.

Характеристика 7: Воспроизводимая методология. Хороший проект является результатом воспроизводимого метода, управляемого требованиями. Такой метод использует четко определенные нотации и приемы, эффективно передающие замысел. Методология должна быть определена в стандартах проектирования.

Характеристика 8: Сопровождаемость. При документировании проекта следует учитывать его последующее сопровождение в целом. Проектирование с учетом сопровождаемости включает создание программного обеспечения, пригодного для повторного использования полностью или частично, для загрузки и для модификации с минимальным воздействием на остальную систему.

Характеристика 9: Робастность. Общую робастность проекта следует учитывать на этапе проектирования и ясно документировать в описании проекта. Примеры факторов робастного проекта включают поведение вне номинального режима, функции прерываний и обработку непредусмотренных прерываний, обработку ошибок и исключений, реакции на отказы,

выявление и удаление непредусмотренной функциональности, потерю питания и восстановление после нее, например холодный и теплый запуск, сбросы, задержку, пропускную способность, полосу пропускания, времена отклика, ограничения ресурсов, обособление ПО, если оно требуется, отключение неиспользуемого кода, если используется отключенный код, и допуски.

Характеристика 10: Документированные проектные решения. Обоснование решений, принятых в ходе проектирования, должно быть документировано. Это позволяет корректно выполнять верификацию и поддерживает сопровождение.

Характеристика 11: Документированные средства обеспечения безопасности. Любое проектное средство, используемое для поддержки безопасности, должно быть четко документировано. Примеры включают сторожевые таймеры, межканальные сравнения, проверки правдоподобия, встроенные процедуры самоконтроля и проверки целостности, например циклический избыточный код или контрольную сумму.

Характеристика 12: Документированные средства информационной безопасности. По мере роста изощренности злоумышленников проект должен предусматривать защиту от эксплуатации уязвимостей и механизмы обнаружения атак.

Характеристика 13: Рассмотрение проекта. На протяжении всего этапа проектирования должны выполняться неформальные технические рассмотрения. Во время таких рассмотрений рецензенты должны содержательно оценивать качество и уместность проекта. Некоторые варианты проекта, или по крайней мере их части, может потребоваться отбросить, чтобы прийти к лучшему решению. Итеративные рассмотрения с участием технически сильных инженеров помогают раньше найти наиболее подходящий проект и тем самым уменьшить число проблем, которые иначе обнаружили бы позже, при формальном рассмотрении проекта или во время тестирования.

Характеристика 14: Тестопригодность. Программное обеспечение должно проектироваться так, чтобы его можно было тестировать. Тестопригодность – это то, насколько легко ПО поддается тестированию. Тестопригодное ПО одновременно наблюдаемо и управляемо. Прессман отмечает, что тестопригодное ПО обладает следующими характеристиками [2]:

- *Работоспособность* – программное обеспечение делает то, что должно делать в соответствии с требованиями и проектом.
- *Наблюдаемость* – входы, внутренние переменные и выходы программного обеспечения могут наблюдаться во время выполнения, чтобы определить, проходит ли тест или нет.
- *Управляемость* – выходы программного обеспечения можно изменять с помощью за-

данных входов.

- *Декомпозируемость* – программное обеспечение построено из компонентов, которые при необходимости можно выделить и тестировать независимо.
- *Простота* – программное обеспечение обладает функциональной, структурной и кодовой простотой. Например, простые алгоритмы тестировать легче, чем сложные.
- *Стабильность* – программное обеспечение не изменяется или изменяется лишь незначительно.
- *Понятность* – хорошая документация помогает специалистам по тестированию понять программное обеспечение и потому проверять его более тщательно.

Некоторые средства, которые могут помочь сделать программное обеспечение тестопригодным, включают следующее [10]:

- *Журналирование ошибок или отказов.* Такая функциональность в программном обеспечении может дать специалистам по тестированию возможность лучше понять поведение ПО.
- *Диагностика.* Диагностическое программное обеспечение, например проверки целостности кода или памяти, может помочь выявлять проблемы в системе.
- *Точки тестирования.* Они предоставляют точки доступа к программному обеспечению, которые могут быть полезны для тестирования.
- *Доступ к интерфейсам.* Это может помочь при интерфейсном и интеграционном тестировании.

Взаимодействие между инженерами по тестированию и проектировщиками может помочь сделать программное обеспечение более тестопригодным. В частности, инженеров по тестированию следует привлекать во время рассмотрений проекта, а по возможности и еще раньше.

Характеристика 15: Отсутствие нежелательных возможностей. Обычно в проектах критического по безопасности программного обеспечения запрещается следующее:

- *Рекурсивное выполнение функций,* то есть функции, которые могут вызывать сами себя прямо или косвенно. Без крайней осторожности и специальных проектных мер использование рекурсивных процедур может привести к непредсказуемому и потенциально большому расходу стековой памяти.
- *Самомодифицирующийся код* – это код, который изменяет собственные инструкции во время выполнения, обычно для сокращения длины пути исполнения инструкций, повышения производительности или уменьшения количества повторяющегося похожего

кода.

- *Динамическое выделение памяти*, если только выделение не выполняется один раз детерминированным образом во время инициализации системы. В DO-332 описаны связанные с динамическим выделением памяти проблемы и приведены рекомендации.

7.4 Верификация проекта

Как и в случае с программными требованиями, проект программного обеспечения необходимо верифицировать. Обычно это выполняется посредством экспертного рассмотрения. Рекомендации по проведению экспертных рассмотрений из главы 6 также применимы к рассмотрению проекта. Во время рассмотрения проекта оцениваются как требования низкого уровня, так и архитектура. Ниже перечислены и пояснены цели DO-178C из таблицы A-4, относящиеся к верификации проекта [1]:

- *Цель 1 таблицы A-4 DO-178C*: «Требования низкого уровня соответствуют требованиям высокого уровня». Это гарантирует, что требования низкого уровня полностью и точно реализуют требования высокого уровня. Иными словами, вся функциональность, определенная в требованиях высокого уровня, отражена в требованиях низкого уровня.
- *Цель 2 таблицы A-4 DO-178C*: «Требования низкого уровня являются точными и согласованными». Это гарантирует, что требования низкого уровня не содержат ошибок и согласованы как между собой, так и с требованиями высокого уровня.
- *Цель 3 таблицы A-4 DO-178C*: «Требования низкого уровня совместимы с целевым вычислителем». Здесь проверяются все зависимости требований низкого уровня от целевой платформы.
- *Цель 4 таблицы A-4 DO-178C*: «Требования низкого уровня поддаются верификации». Обычно здесь внимание сосредоточено на тестопригодности требований низкого уровня. Для уровней A-C требования низкого уровня должны быть протестированы. Поэтому возможность верификации необходимо учитывать уже при первоначальной разработке требований низкого уровня. Дополнительное обсуждение тестопригодности приведено в характеристике 14 раздела 7.3.
- *Цель 5 таблицы A-4 DO-178C*: «Требования низкого уровня соответствуют стандартам». В главе 5 обсуждается разработка стандартов проектирования. Во время рассмотрения требования низкого уровня оцениваются на соответствие этим стандартам.[*]
- *Цель 6 таблицы A-4 DO-178C*: «Требования низкого уровня трассируются к требованиям высокого уровня». Это тесно связано с целью 1 таблицы A-4. Двухнаправленная трассируемость между требованиями высокого уровня и требованиями низкого уровня подтверждает соответствие требований низкого уровня требованиям высокого уровня.

Во время рассмотрения также проверяется точность данных трассировки. Любые неясные связи трассируемости следует проанализировать и либо изменить, либо пояснить в обосновании. Вопросы трассируемости рассматриваются в главе 6.

- *Цель 7 таблицы A-4 DO-178C: «Алгоритмы являются корректными».* Любые математические алгоритмы должен проверять специалист с соответствующей подготовкой, чтобы подтвердить их корректность. Если алгоритм повторно используется из предыдущей системы, где он уже был тщательно проверен и при этом не изменен, то можно использовать свидетельства рассмотрения из предыдущей разработки. Использование таких ранее полученных свидетельств верификации должно быть отмечено в планах.
- *Цель 8 таблицы A-4 DO-178C: «Архитектура программного обеспечения совместима с требованиями высокого уровня».* Часто для подтверждения такой совместимости между требованиями и архитектурой обеспечивают трассируемость или создают отображение.
- *Цель 9 таблицы A-4 DO-178C: «Архитектура программного обеспечения согласована».* Эта цель верификации обеспечивает согласованность и корректность компонентов архитектуры ПО.
- *Цель 10 таблицы A-4 DO-178C: «Архитектура программного обеспечения совместима с целевым вычислителем».* Это подтверждает, что архитектура подходит для конкретной целевой платформы, на которой будет реализовано ПО.
- *Цель 11 таблицы A-4 DO-178C: «Архитектура программного обеспечения поддается верификации».* Как уже отмечалось, тестопригодность следует учитывать при разработке архитектуры. Дополнительное обсуждение тестопригодности приведено в характеристике 14 раздела 7.3.
- *Цель 12 таблицы A-4 DO-178C: «Архитектура программного обеспечения соответствует стандартам».* При разработке архитектуры следует соблюдать стандарты проектирования. Для уровней А, В и С на соответствие стандартам проектирования оцениваются как архитектура, так и требования низкого уровня.
- *Цель 13 таблицы A-4 DO-178C: «Подтверждена целостность обособления ПО».* Если используется обособление ПО, оно должно быть отражено в проекте и верифицировано. Тема обособления ПО рассматривается в главе 21.

Во время рассмотрения проекта инженеры обычно используют контрольный перечень, помогающий провести тщательную оценку проекта. Приведенные ранее рекомендации из раздела 7.3 и цели DO-178C из таблицы A-4 служат хорошей отправной точкой для стандартов проектирования и контрольного перечня рассмотрения.

Литература

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
2. R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 7th edn. (New York: McGraw-Hill, 2010).
3. RTCA DO-248C, *Supporting Information for DO-178C and DO278A* (Washington, DC: RTCA, Inc., December 2011).
4. K. Shumate and M. Keller, *Software Specification and Design: A Disciplined Approach for Real-Time Systems* (New York: John Wiley & Sons, 1992).
5. M. Page-Jones, *The Practical Guide to Structured Systems Design* (New York: Yourdon Press, 1980).
6. J. Cooling, *Software Engineering for Real-Time Systems* (Harlow, U.K.: Addison-Wesley, 2003).
7. H.V. Vliet, *Software Engineering: Principles and Practice*, 3rd edn. (Chichester, U.K.: Wiley, 2008).
8. E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (New York: Yourdon Press, 1975).
9. S. Paasch, Software coupling, presentation at Federal Aviation Administration Designated Engineering Representative Conference (Long Beach, CA: September 1998).
10. C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing* (New York: John Wiley & Sons, 2002).

[*] Скобки добавлены для единообразия.

[*] Следует отметить, что в некоторых проектах к требованиям низкого уровня применяют стандарты требований, а не стандарты проектирования.

Глава 8. Реализация программного обеспечения: кодирование и интеграция

Сокращения

Сокращение	Расшифровка
ACG	Генератор автокода (autocode generator)
ANSI	Американский национальный институт стандартов (American National Standards Institute)
ARM	Справочное руководство по языку Ada (Ada Reference Manual)
CAST	Группа по программному обеспечению сертифицирующих органов (Certification Authorities Software Team)
CPU	Центральный процессор (central processing unit)
CRC	Циклическая избыточная проверка (cyclic redundancy check)
DoD	Министерство обороны США (Department of Defense)
FAQ	Часто задаваемые вопросы (frequently asked question)
I/O	Ввод-вывод (input/output)
IEC	Международная электротехническая комиссия (International Electrical Technical Commission)
ISO	Международная организация по стандартизации (International Standards Organization)
LRM	Справочное руководство по языку (Language Reference Manual)
MISRA	Ассоциация надежности ПО автомобильной промышленности (Motor Industry Software Reliability Association)
PPP	Процесс кодирования на псевдокоде (Pseudocode Programming Process)
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
RTOS	ОСРВ (операционная система реального времени, real-time operating system)
SCI	Указатель конфигурации ПО (Software Configuration Index)
SECI	Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index)
WORA	Написано один раз, работает везде (write once, run anywhere)

8.1 Введение

В этой главе рассматриваются два аспекта реализации программного обеспечения: (1) кодирование и (2) интеграция. Кроме того, обсуждается верификация процессов кодирования и интеграции.

Кодирование – это разработка исходного кода на основе описания проекта. Иногда вместо этого говорят *разработка* или *программирование*, чтобы подчеркнуть: речь не о механическом наборе строк, а о работе, которая требует обдумывания и мастерства, примерно как при строительстве здания или моста. Однако в DO-178C (KT-178C) последовательно используется термин *coding* (кодирование). Логика DO-178C такова: значительная часть построения решения относится скорее к проектированию, чем к кодированию. При этом DO-178C не запрещает проектировщику самому писать код. Кто бы ни определял детали реализации, важность исходного кода нельзя недооценивать. Требования и проект критически важны, но реально летает именно скомпилированный и скомпонованный код.

Интеграция – это получение исполняемого объектного кода с помощью компилятора и компоновщика (линковщика), а затем загрузка этого кода в целевой вычислитель с помощью загрузчика. Интеграцию иногда считают простой технической операцией, но в разработке критичного по безопасности ПО она жизненно важна.

8.2 Кодирование

Кодирование создает исходный код, который затем преобразуется в исполняемый образ и попадает в критичную по безопасности систему. Поэтому это исключительно важный этап разработки программного обеспечения. В этом разделе рассматриваются указания DO-178C по кодированию, распространенные языки, используемые для разработки критичного по безопасности ПО, и рекомендации по кодированию в этой предметной области. Языки C, Ada и ассемблер кратко рассматриваются потому, что именно они наиболее часто применяются во встраиваемых критичных по безопасности системах. Некоторые другие языки тоже кратко упоминаются, но без подробного разбора. Раздел завершается обсуждением двух специальных тем, связанных с кодом: библиотек и генераторов автокода (autocode generators).

Обратите внимание, что тема параметрических или конфигурационных данных тесно связана с кодированием; однако она рассматривается в главе 22, а не в этой главе.

8.2.1 Обзор указаний DO-178C по кодированию

DO-178C содержит указания по планированию, разработке и верификации исходного кода. Во-первых, на этапе планирования разработчик определяет конкретный язык программирования, стандарты кодирования и компилятор согласно разделу 4.4.2 DO-178C. В главе

5 обсуждаются общий процесс планирования и стандарты кодирования. В этой главе приводятся некоторые рекомендации по кодированию, которые следует учесть в стандартах кодирования, специфичных для компании, еще на этапе планирования.

Во-вторых, DO-178C дает указания по разработке кода, см. раздел 5.3 DO-178C. DO-178C объясняет, что основная цель процесса кодирования состоит в разработке исходного кода, который является «трассируемым, верифицируемым, согласованным и корректно реализует требования низкого уровня» [1]. Для достижения этой цели исходный код должен реализовывать описание проекта и только описание проекта, включая требования низкого уровня и архитектуру, трассироваться к требованиям низкого уровня и соответствовать установленным стандартам кодирования. Выходами этапа кодирования являются исходный код и данные трассировки. Инструкции по сборке, включая данные компиляции и компоновки, а также инструкции по загрузке, тоже разрабатываются на этапе кодирования, но рассматриваются как часть интеграции, о которой идет речь в разделе 8.4.

В-третьих, исходный код верифицируется для подтверждения его корректности, согласованности, соответствия проекту, соответствия стандартам и так далее. Таблица А-5 документа DO-178C суммирует цели верификации исходного кода; они рассматриваются в разделе 8.3.

8.2.2 Языки, используемые в критичном по безопасности программном обеспечении

В настоящее время в авиационных системах, критичных по безопасности, чаще всего используются три языка: С, Ada и ассемблер. Есть и унаследованные продукты на других языках, включая FORTRAN и Pascal. C++ применялся в ряде проектов, но обычно с такими жесткими ограничениями, что по сути превращался почти в С. Java и С# использовались в нескольких проектах по разработке инструментальных средств и для этой цели подходят вполне неплохо. Однако сейчас они все еще не готовы для реализации критичных по безопасности систем. Использовались или предлагались и другие языки. Поскольку С, Ada и ассемблер остаются наиболее распространенными, они кратко описаны в следующих подразделах. В таблице 8.1 также приведены сведения о других языках. Подробно обсуждать каждый язык было бы заманчиво, но для этого есть другие хорошие источники.

Таблица 8.1 Языки, которые иногда используются в авиационном программном обеспечении

Язык	Краткая характеристика и применение
C++	C++ был представлен в 1979 году Бьерном Страуструпом в Bell Labs как развитие языка C. Сначала он назывался C with Classes, а в 1983 году получил название C++. В языке сочетаются возможности высокого и низкого уровня. C++ – статически типизированный, свободно форматируемый, многопарадигменный язык общего назначения высокого уровня. Его применяют для прикладных программ, драйверов устройств, встроенного ПО, высокопроизводительных серверных и клиентских приложений, а также для описания аппаратуры. По сравнению с C язык C++ добавляет объектно-ориентированные средства: классы, виртуальные функции, множественное наследование, перегрузку операторов, обработку исключений и шаблоны. Большинство компиляторов C++ также умеют компилировать C. C++ и его подмножество Embedded C++ использовались в нескольких критичных по безопасности системах, но многие объектно-ориентированные возможности при этом не применялись.
C#	C# был представлен в 2001 году командой Microsoft под руководством Андерса Хейлсберга. Изначально C# проектировали как простой, современный и объектно-ориентированный язык общего назначения высокого уровня. Для него характерны строгая типизация, а также императивный, функциональный, декларативный, обобщенный, объектно-ориентированный и компонентно-ориентированный стили. C# применялся для программных инструментов в авиации, но пока не считается достаточно зрелым для критичного по безопасности ПО.
FORTRAN	FORTRAN был представлен в 1957 году Джоном Бэкусом в IBM. Это язык общего назначения высокого уровня, созданный для численных вычислений и научных расчетов. Одна из его важных целей – высокая производительность вычислений. К другим характеристикам относятся процедурный и императивный стили программирования; в более поздних версиях появились программирование массивов, модульное программирование, объектно-ориентированное программирование и обобщенное программирование. Раньше FORTRAN применялся в авионике и до сих пор встречается в отдельных унаследованных системах.
Java	Java был представлен в 1995 году Джеймсом Гослингом в Sun Microsystems. Язык основан на C и C++. Это язык общего назначения высокого уровня, спроектированный так, чтобы как можно меньше зависеть от конкретной реализации. К другим его характеристикам относятся классовая модель, поддержка параллельного выполнения и объектно-ориентированный подход. Java следует принципу "написано один раз, работает везде" (write once, run anywhere): код, работающий на одной платформе, не требует перекомпиляции для запуска на другой. Приложения Java обычно компилируются в байт-код, который хранится в файле класса. Затем этот байт-код может выполняться на любой виртуальной машине Java независимо от архитектуры компьютера. Уже разработана Java для систем реального времени, а подмножество для критичного по безопасности применения находится в работе. Java применялась для некоторых авиационных программных инструментов, но по-прежнему не считается достаточно зрелой для критичного по безопасности ПО.
Pascal	Pascal был представлен в 1970 году Никлаусом Виртом и основан на языке ALGOL. Это императивный и процедурный язык высокого уровня. Он был создан, чтобы поощрять хорошие практики программирования за счет структурного программирования и структурирования данных. Среди дополнительных возможностей Pascal – перечисления, поддиапазоны, записи, динамически выделяемые переменные со связанными указателями, а также множества, с помощью которых программист может задавать сложные структуры данных, например списки, деревья и графы. Pascal строго типизирует все объекты, допускает вложенные определения процедур на любую глубину и позволяет размещать большинство определений и объявлений внутри функций и процедур. Раньше Pascal применялся в авионике и до сих пор встречается в отдельных унаследованных системах.

Источник: Wikipedia, Programming languages, http://en.wikipedia.org/wiki/List_of_programming_languages по состоянию на апрель 2012 года.

8.2.2.1 Язык ассемблера

Язык ассемблера (assembly language) – это низкоуровневый язык программирования для компьютеров, микропроцессоров и микроконтроллеров. Он задает машинный код в символьной форме и поэтому привязан к конкретному центральному процессору (central processing unit, CPU). Обычно язык ассемблера определяется производителем процессора. В отличие от языков высокого уровня, ассемблер не переносится между платформами, хотя переносимость внутри одного семейства процессоров часто возможна. Для перевода операторов ассемблера в машинный код целевого вычислителя используется служебная программа, которая тоже называется ассемблером. Она выполняет взаимно-однозначное отображение инструкций и данных языка ассемблера на машинные инструкции и данные.

Существует два типа ассемблеров: однопроходные и двухпроходные. Однопроходный ассемблер проходит по исходному коду один раз и предполагает, что каждый символ будет определен до первой инструкции, которая на него ссылается. Преимущество однопроходного ассемблера – скорость. Двухпроходный ассемблер на первом проходе создает таблицу всех символов и их значений, а на втором использует ее для генерации кода. Чтобы вычислить адреса символов, ассемблер должен как минимум уметь определить длину каждой инструкции на первом проходе. Преимущество двухпроходного ассемблера в том, что символы можно определять в любом месте исходного текста программы. Это позволяет писать программы более логично и осмысленно; такие программы легче читать и сопровождать [2].

В общем случае ассемблера лучше по возможности избегать. Его трудно сопровождать, типизация данных в нем чрезвычайно слабая, механизмы управления потоком выполнения ограничены или отсутствуют, читать такой код сложно, а переносимость обычно отсутствует. Тем не менее иногда ассемблер необходим: для обработки прерываний, аппаратного тестирования и обнаружения ошибок, взаимодействия с процессором и периферийными устройствами, а также для повышения производительности, например скорости выполнения на критичных участках [3].

8.2.2.2 Ada

Ada впервые появился в 1983 году в версии, известной как Ada-83. На язык повлияли ALGOL и Pascal, а назван он в честь Ады Лавлейс, которую считают первым программистом в истории. Изначально Ada разработали по запросу Министерства обороны США (Department of Defense, DoD), и несколько лет это ведомство предписывало его использовать. До появления Ada в проектах Министерства обороны США применялись буквально сотни языков. Ведомство хотело стандартизировать язык для встроенных систем, систем реального времени и приложений, критичных для выполнения задачи. Ada включает строгую типизацию, пакеты для модульности, проверки во время выполнения, задачи для параллельной обработки, обработку исключений и обобщения. Версии Ada 95 и Ada 2005 добавили объектно-ориентированные возможности. Язык Ada стандартизован в рамках Международной организации по стандартизации (International Standards Organization, ISO), Американского национального института стандартов (American National Standards Institute, ANSI) и Международной электротехнической комиссии (International Electrical Technical Commission, IEC). В отличие от большинства стандартов ISO/IEC, описание языка Ada [*] находится в открытом доступе и бесплатно доступно программистам и разработчикам компиляторов.

Ada обычно предпочитают те, кто особенно серьезно относится к безопасности, потому что язык предоставляет очень сильный набор средств. Он “поддерживает проверки во время вы-

полнения, защищающие от доступа к невыделенной памяти, переполнения буфера, ошибок на единицу, ошибок доступа к массивам и других обнаруживаемых дефектов” [4]. Ada также поддерживает множество проверок во время компиляции. В других языках такие проблемы обнаруживаются только во время выполнения либо требуют явных проверок в исходном коде. В 1997 году Министерство обороны США фактически отказалось от обязательного применения Ada. К тому моменту общее число используемых языков уже сократилось, а оставшиеся языки стали достаточно зрелыми для создания качественных продуктов. Иными словами, вместо сотен незрелых языков в распоряжении уже было несколько зрелых.

8.2.2.3 C

C – один из самых популярных языков программирования в истории. Изначально его разработал Деннис Ритчи в Bell Telephone Laboratories в 1969-1973 годах для использования вместе с операционной системой UNIX. К 1973 году язык стал настолько мощным, что большая часть ядра UNIX была переписана на C. Благодаря этому UNIX стала одной из первых операционных систем, реализованных не на ассемблере [5]. C предоставляет низкоуровневый доступ к памяти и содержит конструкции, хорошо отображающиеся на машинные инструкции. Поэтому ему нужна минимальная поддержка во время выполнения, и он удобен для приложений, которые раньше писались на ассемблере.

Некоторые не готовы называть C языком высокого уровня и предпочитают термин “язык среднего уровня”. У него действительно есть свойства языков высокого уровня, такие как структурированные данные, структурированное управление потоком, машинная независимость и операторы [6]. Однако есть и низкоуровневые конструкции, например побитовые операции. В C весь исполняемый код размещается внутри функций. C имеет слабую типизацию, допускает сокрытие переменных во вложенных блоках, позволяет обращаться к памяти компьютера через указатели, содержит сравнительно небольшой набор зарезервированных ключевых слов, использует библиотечные процедуры для сложной функциональности, например ввода-вывода (input/output, I/O), математических функций и обработки строк, а также применяет составные операторы, такие как +=, -=, *=, ++ и другие. Текст программы на C имеет свободный формат, а оператор завершается точкой с запятой.

C – мощный язык. Но вместе с этой мощностью появляется необходимость пользоваться им с крайней осторожностью. Стандарт MISRA-C [7], разработанный Ассоциацией надежности программного обеспечения автомобильной промышленности (Motor Industry Software Reliability Association), содержит отличные рекомендации по безопасному применению C и часто используется как основа для корпоративных стандартов кодирования.

8.2.3 Выбор языка и компилятора

При выборе языка и компилятора для критичного по безопасности проекта или группы проектов следует учитывать несколько аспектов.

Соображение 1: возможности языка и компилятора. Язык и компилятор должны справляться со своей задачей. Однажды я работала с компанией, которая разработала компилятор для безопасного подмножества Ada. Компания даже потратила силы и деньги, чтобы квалифицировать его как средство разработки уровня А. Однако подмножество Ada оказалось слишком узким для реальных проектов, поэтому отрасль им не воспользовалась. Ниже перечислены базовые возможности языка, важные для большинства проектов:

- читаемость кода: чувствительность к регистру и смещение регистров, понятность зарезервированных слов и математических символов, гибкие правила форматирования и ясные соглашения об именовании;
- способность обнаруживать ошибки на этапе компиляции, например опечатки и типичные ошибки кодирования;
- способность обнаруживать ошибки во время выполнения, включая исчерпание памяти, конструкции обработки исключений и арифметические ошибки, например переполнение чисел, выход за границы массива и деление на ноль [3];
- переносимость между платформами, подробнее об этом говорится в соображении 9;
- поддержка модульности, включая инкапсуляцию и сокрытие информации;
- строгая типизация данных;
- четко определенные управляющие структуры;
- поддержка взаимодействия с операционной системой реального времени (real-time operating system, RTOS), если такая система используется в текущем или будущих проектах;
- поддержка потребностей систем реального времени, например многозадачности и обработки исключений;
- взаимодействие с другими языками, например с ассемблером; возможность отдельной компиляции и отладки;
- взаимодействие с аппаратурой, если операционная система реального времени не используется.

Соображение 2: критичность программного обеспечения. Чем выше критичность, тем более управляемым и предсказуемым должен быть язык. Проект уровня D, возможно, сможет пройти сертификацию с Java или C#, но для проектов уровня А требуется более высокая степень детерминированности и зрелости языка. Некоторые производители компиляторов предлагают для своего языка общего назначения подмножество, рассчитанное на системы

реального времени и/или критичное по безопасности ПО.

Соображение 3: опыт персонала. Программисты склонны мыслить на языке, который знают лучше всего. Если инженер является экспертом по Ada, ему будет трудно переключиться на С. Аналогично программирование на ассемблере требует особых навыков. Программисты могут работать на нескольких языках, но нужно время, чтобы уверенно владеть каждым из них и хорошо понимать как возможности, так и подводные камни.

Соображение 4: поддержка безопасности в языке. Язык и компилятор должны позволять выполнить применимые цели DO-178С, а также обязательные требования безопасности. Для проектов уровня А требуется верифицировать выход компилятора, чтобы доказать, что он не генерирует непредусмотренный код. Именно поэтому организации обычно выбирают зрелые, стабильные и хорошо зарекомендовавшие себя компиляторы.

Соображение 5: инструментальная поддержка языка. Для разработки нужны подходящие инструменты. Следует учитывать как минимум следующее:

- компилятор должен обнаруживать ошибки и поддерживать потребности, связанные с безопасностью; также необходим надежный компоновщик;
- важен хороший отладчик; он должен быть совместим с выбранным целевым вычислителем;
- следует учитывать тестопригодность языка; средства тестирования и анализа часто зависят от языка, а иногда даже от конкретного компилятора.

Соображение 6: совместимость интерфейсов с другими языками. В большинстве проектов используется по меньшей мере один язык высокого уровня и ассемблер. На одном проекте, в котором я участвовала, для бортового ПО использовались Ada, С и ассемблер, а для инструментов – С++, Java и С#. Выбранные язык и компилятор должны позволять взаимодействовать с ассемблером и кодом на других используемых языках. Обычно файлы ассемблера компонуют вместе со скомпилированным кодом языка высокого уровня. Джим Кулинг удачно формулирует это так: “Одна из крайне желательных возможностей – способность ссылаться из кода на ассемблере на обозначения языка высокого уровня (например, переменные) и наоборот. Для профессиональной работы это должно быть обязательным. Важна также степень перекрестной проверки, которую компоновщик выполняет между подпрограммами языка высокого уровня и ассемблерными подпрограммами (например, по номерам версий)” [3].

Соображение 7: репутация компилятора. Обычно желательно использовать компилятор, совместимый с международными стандартами, например стандартами Американского национального института стандартов (ANSI) или Международной организации по стандар-

тизации (ISO). Даже если используется только подмножество возможностей компилятора, важно убедиться, что применяемые функции корректно реализуют язык. Большинство разработчиков критичного по безопасности ПО выбирают зрелый компилятор с проверенной репутацией. Для проектов уровня А необходимо показать, что код, сгенерированный компилятором, согласован с исходным кодом. Не каждый компилятор способен удовлетворить этим критериям.

Соображение 8: совместимость с выбранной целевой платформой. Большинство компиляторов рассчитаны на конкретную целевую платформу. Выбранные компилятор и среда должны генерировать код, совместимый с используемым процессором и периферийными устройствами. Обычно требуются следующие возможности доступа к процессору и устройствам: управление памятью для данных, кода, кучи и стека, интерфейс и управление периферией, обработка прерываний и поддержка специальных машинных операций [3].

Соображение 9: переносимость на другие целевые платформы. При выборе языка и компилятора большинство компаний учитывает переносимость кода. Значительная часть авиационных проектов опирается на уже существующее программное обеспечение или системы, а не начинается с полностью нового кода. Со временем процессоры становятся мощнее, а старые устаревают. Поэтому важно выбирать язык и компилятор, которые хотя бы в некоторой степени допускают перенос на другие целевые платформы. Это не всегда можно точно предсказать, но учитывать это необходимо. Например, ассемблер обычно непереносим, поэтому при каждой смене процессора код приходится изменять. Если используется то же семейство процессоров, изменения могут быть минимальными, но учитывать их все равно нужно. Ada и С обычно более переносимы, хотя и у них могут быть зависимости от целевой платформы. Java проектировался как хорошо переносимый язык, но, как уже обсуждалось, сейчас он еще не готов к применению в критичной по безопасности области.

8.2.4 Общие рекомендации по кодированию

Этот раздел не претендует на роль исчерпывающего руководства по кодированию. Вместо этого он дает общий обзор практик написания кода для критичного по безопасности программного обеспечения. Обзор основан на тысячах строк кода на разных языках и в разных компаниях, которые мне приходилось анализировать. Эти рекомендации применимы к любому языку и могут учитываться в корпоративных стандартах кодирования.

Рекомендация 1: используйте хорошие методы проектирования. Проект – это чертеж для кода. Поэтому хороший проект важен для получения хорошего программного обеспечения. Не следует ожидать, что разработчик компенсирует недостатки требований и проекта на этапе кодирования. В главе 7 перечислены признаки хорошего проекта: например, слабая межкомпонентная связность, сильная внутренняя связность компонентов, абстракция и мо-

дульность.

Рекомендация 2: поощряйте хорошие практики кодирования. Хорошие программисты гордятся тем, что умеют делать то, что многие считают невозможным; маленькие чудеса у них случаются почти регулярно. Но программисты – народ своеобразный. Один инженер, с которым я работала, проводил вечера за чтением книг по алгоритмам. Я и сама не чужда техническим странностям, и дом у меня забит книгами, но книги по алгоритмам не кажутся мне *настолько* увлекательными. Ниже приведены рекомендации, которые помогают развивать хорошие практики кодирования:

- *поощряйте командную работу.* Командная работа помогает отсеивать плохие практики и нечитаемый код. Реализовать ее можно по-разному: парным кодированием, неформальными ежедневными или еженедельными рассмотрениями кода либо схемой наставник-стажер;
- *проводите разборы кода.* Формальные рассмотрения фактически обязательны для более высоких уровней программного обеспечения. Но большую пользу дают и менее формальные рассмотрения кода небольшими группами программистов. В общем случае полезно, чтобы каждую строку кода рассмотрели как минимум еще два программиста. У этого процесса есть несколько достоинств. Во-первых, он создает здоровую конкуренцию среди коллег: никому не хочется выглядеть глупо перед своими. Во-вторых, рассмотрения помогают унифицировать практики кодирования. В-третьих, они поддерживают непрерывное улучшение: увидев, как кто-то решил задачу, программист может взять этот прием в собственный арсенал. В-четвертых, рассмотрения могут повышать повторное использование кода: некоторые подпрограммы могут пригодиться в нескольких функциях. Наконец, рассмотрения обеспечивают преемственность, если кто-то покидает команду;
- *предоставляйте хорошие примеры кода.* Они могут служить учебным пособием для команды. Некоторые компании ведут перечень лучших фрагментов кода и пополняют его удачными примерами. Это дает обучающий материал и одновременно стимулирует программистов писать хороший код, который может попасть в такой перечень;
- *требуйте соблюдения стандарта кодирования.* DO-178C требует соответствия стандартам кодирования для уровней А-С. Слишком часто стандарты игнорируются до момента формального рассмотрения кода, когда вносить изменения уже тяжело. Программисты должны быть обучены этим стандартам и обязаны им следовать. Это, в свою очередь, означает, что сами стандарты должны быть разумными и применимыми на практике;
- *делайте код доступным всей команде.* Когда люди знают, что их работу будут видеть

другие, они заметно чаще поддерживают ее в аккуратном состоянии;

- *поощряйте хороший код.* Отмечайте тех, кто пишет хороший код. Оценка качества кода часто опирается на отзывы коллег, число дефектов, найденных при рассмотрении, скорость разработки, а также общую стабильность и зрелость кода по ходу проекта. Награда должна быть такой, которая действительно имеет ценность для команды. При этом с системой вознаграждений нужно обращаться осторожно: люди склонны оптимизировать именно то, что измеряется. Например, если оценивать работу по числу написанных строк, это вполне может поощрять неэффективное кодирование;
- *поощряйте команду брать ответственность за свою работу.* Каждый участник команды должен отвечать за свой результат. Нужно уметь признавать ошибки и не скатываться в поиск виноватых. Оправдания вредят команде. Вместо оправданий лучше предлагать решения [8];
- *создавайте возможности для профессионального развития.* Поддерживайте обучение и все, что способствует профессиональному росту программистов.

Рекомендация 3: не допускайте деградации программного обеспечения. Деградация ПО происходит тогда, когда в нем нарастает беспорядок. Переход от чистого кода к запутанному и некорректному начинается медленно. Сначала кто-то решает добавить комментарии *потом*; затем вставляет пробный код, чтобы посмотреть, как он работает, и забывает удалить; и так далее. Вскоре появляется набор кода, который невозможно ни читать, ни исправлять. Деградация кода начинается с пренебрежения деталями. Лечится это только упреждающими действиями. Если что-то выглядит неправильно, этим нужно заниматься, а не игнорировать. Если времени на исправление нет, следует хотя бы вести упорядоченный список проблем и закрыть их до рассмотрения кода.

Рекомендация 4: постоянно держите в уме сопровождаемость. Стив Макконнелл пишет: «Помните о человеке, которому придется модифицировать ваш код. Программирование – это в первую очередь общение с другим программистом и только во вторую – с компьютером» [6]. Большая часть времени разработчика уходит на рассмотрение и изменение кода, будь то в исходном проекте или при сопровождении. Поэтому код нужно писать для людей, а не только для машин. Многие советы из этого раздела помогают именно сопровождаемости.

Читаемость и хорошо организованная структура программы должны быть приоритетом. Код со слабой межкомпонентной связностью тоже упрощает сопровождение.

Рекомендация 5: продумывайте код заранее. На каждую главу этой книги я тратила часы на изучение темы и организацию мыслей в виде плана. После этого собственно написание

текста идет довольно быстро. С кодом все точно так же.

По моему опыту, самые проблемные фрагменты кода появляются тогда, когда программистов заставляют писать быстро. Требования и проект либо сырые, либо вообще отсутствуют, и программисты просто штампуют код. Это похоже на строительство дома без чертежа и из сомнительных материалов: он обязательно развалится, вопрос только в том, когда именно.

Хороший проектный документ дает программистам надежную отправную точку. Но очень немногие способны сразу перейти от проекта к коду без какого-либо промежуточного шага. Этот шаг может быть формальным или неформальным. Он может стать частью проекта или превратиться в комментарии в самом коде.

Макконнелл рекомендует процесс кодирования на псевдокоде (Pseudocode Programming Process, PPP) [6]. В этом процессе псевдокод используется для проектирования и рассмотрения подпрограммы еще до ее кодирования, рассмотрения и тестирования. В PPP конкретные операции подпрограммы описываются формулировками, похожими на обычный английский текст. Эти формулировки не зависят от языка, поэтому псевдокод можно реализовать на любом языке, а сам псевдокод остается на более высоком уровне абстракции, чем собственно код. Он передает замысел, а не реализацию на целевом языке. Псевдокод может стать частью детального проекта, и его зачастую легче рассматривать для оценки корректности, чем сам код. Интересный плюс этого процесса состоит в том, что псевдокод может стать основой для исходного кода и затем поддерживаться в виде комментариев в коде. Кроме того, псевдокод проще обновлять, чем сам код. Подробное описание PPP см. в главе 9 книги Стива Макконнелла *Code Complete*. Это лишь один из многих возможных подходов к созданию подпрограмм или классов, но он может быть весьма эффективным. Как отмечалось в главе 7, использование псевдокода в качестве требований низкого уровня к ПО (ТНУ) может порождать некоторые сертификационные сложности, которые обсуждаются в разделе часто задаваемых вопросов (frequently asked questions) документа DO-248C, в вопросе 82. Краткое изложение этих проблем см. в разделе 7.1.2.

Рекомендация 6: делайте код читаемым. Читаемость кода исключительно важна. Она влияет на способность понимать код, рассматривать его и сопровождать. Она помогает рецензентам понять код и убедиться, что он удовлетворяет требованиям. Она также критически важна для сопровождения кода еще долго после того, как программист перешел на другой проект.

Однажды меня попросили провести рассмотрение кода системы связи. Я прочитала требования и проект по одной конкретной функции и стала продираться через код. Код на С был очень хитроумным и лаконичным. Комментарии не было, пробельные разделители почти отсутствовали. Было чрезвычайно трудно понять, что делает этот код и как он соотносится

с требованиями. Поэтому в конце концов я попросила программиста объяснить его. Ему понадобилось некоторое время, чтобы вспомнить свой замысел, но в итоге он смог провести меня по коду. После того как я поняла ход его мысли, я сказала примерно так: «Код очень трудно читать». На что он без малейшей паузы ответил: «Мне его тоже было трудно писать, вот я и решил, что другим тоже должно быть трудно его читать». Не самые мудрые слова, которые стоит говорить аудитору Федерального управления гражданской авиации США, но по крайней мере человек был честен. И, пожалуй, он очень точно выразил образ мышления многих программистов.

Некоторые программисты считают, что писать малопонятный код – это их *гарантия занятости*. На деле же это проблема для компании. Невозможно слишком часто повторять: код должен быть написан для людей не меньше, чем для компьютера. Макконнелл пишет: «Меньшая часть работы программиста состоит в том, чтобы написать программу так, чтобы ее мог читать компьютер; большая часть – в том, чтобы ее могли читать другие люди» [6].

Стремление сделать код читаемым влияет на его понятность, удобство рассмотрения, уровень ошибок, отлаживаемость, изменяемость, качество и стоимость [6]. Для реальных проектов все это имеет значение.

На читаемость особенно сильно влияют два аспекта кодирования: (1) оформление и (2) комментарии. Ниже приведено краткое изложение рекомендаций по оформлению и комментариям, которое следует считать хотя бы минимальным набором. В разделе «Рекомендуемая литература» в конце главы приведены и другие полезные источники.

1. Рекомендации по оформлению кода

- a. *Показывайте логическую структуру кода.* Как общее правило, операторы следует располагать с отступом под тем оператором, которому они логически подчинены. Исследования показывают, что отступы улучшают понимание. Обычно предпочтительны отступы от двух до четырех пробелов, поскольку больший размер уже ухудшает восприятие [6].
- b. *Щедро используйте пробельные разделители.* Сюда относятся пустые строки между блоками кода или группами связанных операторов, а также отступы, показывающие логическую структуру кода. Например, при написании подпрограммы полезно отделять пустыми строками заголовок подпрограммы, объявления данных и тело.
- c. *Учитывайте изменяемость при выборе правил оформления.* Определяя стиль и правила оформления, выбирайте то, что легко поддерживать. Некоторые программисты любят, например, украшать заголовки фиксированным числом звездочек,

но на их правку потом уходит лишнее время и силы. Программисты склонны игнорировать то, что бессмысленно тратит их время, поэтому практики оформления должны быть практичными.

- d. *Держите тесно связанные элементы вместе.* Например, если оператор переносится на следующую строку, переносите его в читаемом месте и делайте отступ второй строки, чтобы было очевидно, что это одна конструкция. Точно так же следует держать вместе и связанные операторы.
- e. *Используйте только один оператор в строке.* Многие языки допускают несколько операторов в одной строке, но это ухудшает читаемость. Один оператор на строку повышает читаемость, упрощает оценку сложности и помогает выявлению ошибок.
- f. *Ограничивайте длину операторов.* В общем случае строки не должны превышать 80 символов. Более длинные строки трудно читать. Это не жесткое правило, а общая рекомендованная практика для повышения читаемости.
- g. *Делайте объявления данных понятными.* Для ясного объявления данных рекомендуется следующее: одно объявление на строку, объявление переменных рядом с местом их первого использования и упорядочение объявлений по типам [6].
- h. *Используйте круглые скобки.* Скобки помогают прояснить выражения, в которых участвуют более двух членов.

2. Рекомендации по комментариям в коде

- Комментарии в коде часто отсутствуют, бывают неточными или бесполезными. Ниже приведены рекомендации, помогающие комментировать код действительно эффективно.
- a. *Комментируйте «почему».* Как правило, комментарии должны объяснять, *почему* что-то делается, а не *как* это делается. Комментарии должны кратко излагать назначение и цель, то есть замысел кода. Сам код обычно показывает, как именно это сделано. Используйте комментарии, чтобы подготовить читателя к тому, что последует дальше. Обычно для каждого блока кода достаточно одного-двух предложений.
- b. *Не комментируйте очевидное.* Хороший код должен быть самодокументируемым, то есть читаемым без дополнительных объяснений. «Для многих хорошо написанных программ код сам по себе является лучшей документацией» [9]. Комментарии следует использовать для того, что из кода неочевидно, например для назначения и целей. Точно так же бесполезны комментарии, которые просто повторяют сам код.
- c. *Комментируйте подпрограммы.* Используйте комментарии для объяснения назна-

чения подпрограмм, их входов и выходов, а также любых важных предположений. Для подпрограмм желательно располагать комментарии рядом с тем кодом, который они описывают. Если все комментарии сосредоточены в начале подпрограммы, сам код все равно может оставаться трудночитаемым, а комментарии могут перестать обновляться вместе с кодом. Предпочтительно кратко пояснить подпрограмму в начале, а затем включать более конкретные комментарии по ходу ее тела. Если подпрограмма изменяет глобальные данные, это особенно важно пояснить.

- d. *Используйте комментарии для глобальных данных.* Глобальные данные следует применять осторожно, но если они используются, их нужно комментировать, чтобы обеспечить корректное использование. Комментируйте глобальные данные в месте их объявления, указывая назначение этих данных и причину, по которой они нужны. Некоторые разработчики даже вводят отдельное соглашение об именовании глобальных данных, например начинают переменные с `g_`. Если отдельного соглашения об именовании нет, комментарии становятся особенно важны для правильного использования глобальных данных [6].
- e. *Не используйте комментарии как компенсацию плохого кода.* Плохой код нужно не объяснять, а избегать его.
- f. *Комментарии должны соответствовать коду.* Если код обновляется, комментарии тоже должны обновляться.
- g. *Документируйте все предположения.* Программист должен документировать все делаемые предположения и явно пометать их как предположения.
- h. *Документируйте все, что может вызвать удивление.* Неясный код следует оценить и решить, не лучше ли его переписать. Если код все же приемлем, следует добавить комментарий, объясняющий его обоснование. Например, иногда ограничения по производительности подталкивают к особенно *хитрому* коду. Но это должно быть исключением, а не правилом. Такой код следует пояснять комментарием.
- i. *Выравнивайте комментарии относительно соответствующего кода.* Каждый комментарий должен быть расположен рядом с тем кодом, который он объясняет.
- j. *Избегайте комментариев в конце строки.* В общем случае комментарии лучше выносить на отдельную строку, а не дописывать в конец строки кода. Возможные исключения – объявления данных и пометки о конце блока для длинных участков кода [6].
- k. *Перед каждым комментарием оставляйте пустую строку.* Это улучшает общую читаемость.

- l. *Пишите комментарии так, чтобы их было легко сопровождать.* Комментарии должны писаться в стиле, удобном для сопровождения. Иногда в попытке сделать код красивым его одновременно делают неудобным для поддержки. Программисты не должны тратить драгоценное время на подсчет дефисов и выравнивание звездочек.
- m. *Комментируйте упреждающе.* Комментирование должно быть неотъемлемой частью процесса кодирования. При правильном использовании оно может даже стать планом кода, по которому программист затем организует сам код. Стоит помнить: если код трудно комментировать, возможно, он изначально не очень хорош и его нужно изменить.
- n. *Не перегружайте код комментариями.* Как и слишком малое число комментариев, слишком большое число тоже вредно. Я редко вижу избыточно прокомментированный код, но когда вижу, это обычно следствие ненужного дублирования. Комментарии не должны просто повторять код, а должны объяснять, зачем этот код нужен. Исследования IBM показали, что пик ясности достигается примерно при одном комментарии на каждые десять операторов. Больше или меньшее количество комментариев снижало понятность [10]. Разумеется, это лишь общее правило, но его стоит иметь в виду. Не следует чрезмерно концентрироваться на числе комментариев, лучше оценивать, объясняют ли они, зачем существует данный код.

Рекомендация 7: управляйте сложностью и уменьшайте ее. Управление сложностью и ее снижение – одна из важнейших технических тем в разработке программного обеспечения. Работа со сложностью начинается на уровне требований и проекта, но сложность кода тоже нужно внимательно отслеживать и контролировать.

Когда код становится слишком сложным, он делается нестабильным и неуправляемым, а производительность разработки начинает двигаться в отрицательную сторону.

На одном особенно трудном проекте мне действительно пришлось читать запутанный код, чтобы понять, что вообще делает программное обеспечение. Требования были бесполезны, проект отсутствовал, а код был собран на скорую руку. Понадобились годы, чтобы привести код в порядок, восстановить требования и проект и доказать, что программа действительно делает то, что должна делать. Сложность была лишь одной из множества проблем на этом проекте.

Чтобы избежать такого сценария, приведу несколько советов, помогающих управлять сложностью и уменьшать ее:

- используйте модульность, чтобы разбивать задачу на более мелкие и управляемые ча-

сти;

- уменьшайте межкомпонентную связность между подпрограммами;
- используйте перегрузку операторов и имен переменных с осторожностью либо по возможности вовсе запрещайте ее;
- используйте единственную точку входа и выхода для подпрограмм;
- делайте подпрограммы короткими и связными;
- применяйте логичные и понятные соглашения об именовании;
- уменьшайте количество вложенных решений, а также глубину дерева наследования; избегайте слишком хитрых подпрограмм, которые трудно понять; вместо этого стремитесь к простым и понятным решениям;
- если возможно, преобразуйте вложенный `if` в набор конструкций `if - then - else` или в оператор `case`;
- используйте средство или метод измерения сложности, чтобы выявлять чрезмерно сложный код; затем перерабатывайте код по мере необходимости;
- давайте код на рассмотрение людям, которые с ним не знакомы, чтобы убедиться, что он понятен; очень легко настолько увязнуть в деталях, что теряется общая картина; независимые рассмотрения технически компетентными специалистами дают ценную проверку здравого смысла;
- используйте модульность и сокрытие информации, чтобы отделять детали низкого уровня от использования модуля; это по сути позволяет разделить задачу на более мелкие модули, подпрограммы, компоненты или классы и скрыть сложность, чтобы с ней не приходилось сталкиваться каждый раз; у каждого модуля должен быть четко определенный интерфейс и тело; в теле находится реализация модуля, а интерфейс – это то, что видит пользователь;
- по возможности минимизируйте использование обработки по прерываниям и многозадачности;
- ограничивайте размер файлов; магического числа тут нет, но в общем случае файлы объемом более 200-250 строк кода становятся трудными для чтения и сопровождения.

Рекомендация 8: практикуйте защитное программирование. В документе DO-248C, в вопросе 32 раздела часто задаваемых вопросов, говорится: «Приемы защитного программирования – это методы, которые могут использоваться для предотвращения выполнения кодом непреднамеренных или непредсказуемых операций за счет ограничения применения структур, конструкций и приемов, способных вносить эксплуатационные отказы и ошибки в код»

[11]. Далее DO-248C рекомендует в процессе кодирования избегать ошибок входных данных, недетерминизма, избыточной сложности, интерфейсных ошибок и логических ошибок. Защитное программирование повышает общую устойчивость кода, помогая избегать нежелательных результатов и защищаться от непредсказуемых событий.

Рекомендация 9: обеспечивайте детерминированность программного обеспечения. Критичное по безопасности ПО должно быть детерминированным. Поэтому практик кодирования, которые могут приводить к недетерминизму, следует избегать или очень жестко их контролировать: например, саомодифицирующегося кода, динамического выделения и освобождения памяти, динамического связывания, чрезмерного использования указателей, множественного наследования или полиморфизма. Детерминированности также помогают хорошо определенные языки, проверенные компиляторы, ограниченная оптимизация и ограниченная сложность.

Рекомендация 10: упреждающе работайте с типовыми ошибками. Полезно вести журнал типовых ошибок, который помогает обучению и повышает осведомленность всей команды, а также содержит рекомендации, как таких ошибок избегать. Например, интерфейсные ошибки встречаются часто. С ними можно бороться, уменьшая сложность интерфейсов, последовательно используя единицы измерения и точность, минимизируя использование глобальных переменных и применяя утверждения (assertions), чтобы выявлять несовпадающие предположения об интерфейсах. Другой пример – типовые ошибки логики и вычислений. Их можно избегать, анализируя вопросы точности и преобразований, например масштабирование фиксированной точки, отслеживая ошибки в числе итераций циклов и используя адекватную точность чисел с плавающей точкой.

В своей книге *The Art of Software Testing* Майерс перечисляет 67 распространенных ошибок кодирования и разбивает их на следующие категории: ошибки обращения к данным, ошибки объявления данных, вычислительные ошибки, ошибки сравнения, ошибки потока управления, интерфейсные ошибки, ошибки ввода-вывода и прочие ошибки [12]. Аналогично, в книге *Testing Computer Software* Кем Канер и соавторы выделяют и классифицируют 480 распространенных дефектов программного обеспечения [13]. Эти источники дают отправную точку для перечня типовых ошибок, но только отправную точку. Набор типовых ошибок будет зависеть от используемого языка, опыта команды, типа системы и так далее.

Рекомендация 11: используйте утверждения во время разработки. Утверждения можно использовать для проверки условий, которые никогда не должны возникать, тогда как код обработки ошибок применяется для условий, которые все же могут возникнуть. Ниже приведены несколько рекомендаций по использованию утверждений [6,8]:

- утверждения не должны включать исполняемый код, поскольку обычно они отключа-

ются на этапе компиляции;

- если кажется, что нечто не может случиться никогда, используйте утверждение (assertion, assert-проверку), чтобы проверить, что этого действительно не происходит; утверждения полезны для проверки предусловий и постусловий; при этом они не должны подменять собой полноценную обработку ошибок.

Рекомендация 12: реализуйте обработку ошибок. Обработка ошибок похожа на утверждения, но применяется для условий, которые реально могут возникнуть. Обработка ошибок проверяет некорректные входные данные, тогда как утверждения проверяют неправильный код, то есть дефекты [6]. Данные не всегда поступают в корректном формате или с допустимыми значениями, поэтому необходимо защищаться от недопустимого входа. Например, следует проверять значения из внешних источников на допустимость диапазона и наличие повреждения данных, отслеживать переполнения буферов и целочисленные переполнения, а также проверять значения входных параметров подпрограмм [11].

На ошибку можно реагировать по-разному: вернуть нейтральное значение, подставить следующий корректный элемент данных, вернуть тот же ответ, что и в предыдущий раз, подставить ближайшее допустимое значение, записать предупреждение в файл, вернуть код ошибки, вызвать подпрограмму обработки ошибки, вывести сообщение об ошибке, обработать ошибку локально или завершить работу системы [6]. Очевидно, конкретная реакция зависит от критичности программного обеспечения и общей архитектуры. Важно обрабатывать ошибки единообразно по всей программе. Кроме того, нужно убедиться, что код верхнего уровня действительно обрабатывает ошибки, о которых сообщает код нижнего уровня [6]. Например, приложение, то есть код верхнего уровня, должно обрабатывать ошибки, сообщаемые операционной системой реального времени, то есть кодом нижнего уровня.

Рекомендация 13: реализуйте обработку исключений. Исключения – это ошибки или отказовые состояния, при которых дальнейшее выполнение программы теряет смысл [3]. При генерации исключения должна вызываться подпрограмма его обработки. Обработка исключений – один из наиболее эффективных способов борьбы с проблемами времени выполнения [3]. Однако языки различаются по тому, как именно они реализуют исключения, а в некоторых конструкция исключений вовсе отсутствует. Если исключения не являются частью языка, программисту придется самостоятельно реализовывать проверки ошибочных состояний в коде. Ниже приведены несколько советов по использованию исключений [6,8]:

- программа должна генерировать исключения только для действительно исключительных ситуаций, то есть таких, которые нельзя обработать другими приемами кодирования;
- исключения должны уведомлять другие части программы об ошибках, требующих ре-

акции;

- по возможности ошибочные состояния следует обрабатывать локально, а не просто передавать дальше;
- исключения следует генерировать на правильном уровне абстракции;
- сообщение об исключении должно указывать, какие именно данные привели к его возникновению;
- программисты должны знать, какие исключения могут генерировать используемые библиотеки и подпрограммы.

В проекте должен быть установлен стандартный подход к использованию исключений.

Рекомендация 14: применяйте общие практики кодирования для подпрограмм, использования переменных, условных операторов, циклов и управления. Стандарты кодирования должны определять рекомендуемые практики для подпрограмм, например функций или процедур, для использования переменных, например соглашений об именовании и применения глобальных данных, для условных операторов, циклов управления и общих вопросов управления, например рекурсии, использования goto, глубины вложенности и сложности.[*]

Рекомендация 15: избегайте известных источников проблем. Составить исчерпывающий перечень всего, что может пойти не так в процессе кодирования, невозможно. Однако есть несколько хорошо известных источников неприятностей и практики, которые помогают их избежать.

1. *Минимизируйте использование указателей.* Указатели – одна из самых подверженных ошибкам областей кодирования. Я даже не стану описывать, сколько часов мне приходилось тратить на поиск проблем, связанных с указателями, когда над головой висел совершенно безумный срок сдачи. Некоторые компании предпочитают вовсе избегать указателей. Если удастся найти разумный способ обойтись без них, это определенно рекомендуется. Но такая возможность есть не всегда. Если указатели все же используются, их применение следует свести к минимуму и обращаться с ними очень осторожно.[†]
2. *Ограничивайте использование наследования и полиморфизма.* Эти темы относятся к объектно-ориентированной технологии, которая рассматривается в главе 15.
3. *Будьте осторожны с динамическим выделением памяти.* На большинстве сертификационных проектов динамическое выделение памяти запрещено. DO-332, то есть дополнение по объектно-ориентированным технологиям, дает несколько рекомендаций по его безопасному применению, если оно все же используется.

4. *Минимизируйте межкомпонентную связность и максимизируйте внутреннюю связность компонентов.* Как отмечалось в главе 7, желательно уменьшать межкомпонентную связность между компонентами кода и создавать компоненты с высокой внутренней связностью каждого компонента. Проект закладывает основу для этого, но реализуют его именно программисты, так что это стоит подчеркнуть еще раз. Эндрю Хант и Дэвид Томас рекомендуют: пишите изолированный код, то есть создавайте модули, которые не раскрывают другим модулям ничего лишнего и не зависят от деталей реализации других модулей [8]. Это и есть идея кода со слабой межкомпонентной связностью, то есть развязанного кода. Чтобы уменьшить межкомпонентную связность, ограничьте взаимодействие модулей; когда такое взаимодействие необходимо, должно быть ясно, зачем оно нужно и как именно осуществляется.
5. *Минимизируйте использование глобальных данных.* Глобальные данные доступны всем частям проекта и потому могут изменяться любым участником по мере развития работы. Как уже отмечалось, глобальные данные следует сводить к минимуму, чтобы поддерживать слабую межкомпонентную связность и делать код более детерминированным. Среди потенциальных проблем глобальных данных можно отметить непреднамеренное изменение значений, ухудшение повторного использования кода, неопределенность порядка инициализации и снижение модульности программы. Глобальные данные следует использовать только тогда, когда это действительно необходимо. Если они используются, полезно каким-то образом их отличать, например с помощью соглашения об именовании. Кроме того, полезно реализовать некоторый механизм блокировки или защиты доступа к глобальным данным. Также важен точный и актуальный словарь данных, содержащий имена глобальных данных, описания, типы, единицы измерения, а также перечни тех, кто эти данные читает и записывает. Точная документация глобальных данных будет критически важна при анализе межкомпонентной связности по данным, который рассматривается в главе 9.
6. *Используйте рекурсию осторожно или не используйте совсем.* Некоторые стандарты кодирования полностью запрещают рекурсию, особенно для более критичных уровней программного обеспечения. Если рекурсия все же применяется, возможно для менее критичных уровней, в проекте должны быть предусмотрены явные механизмы защиты от переполнения стека из-за неограниченной рекурсии. Раздел 6.3.3.d DO-178C требует предотвращать «неограниченные рекурсивные алгоритмы» [1]. В вопросе 39 раздела часто задаваемых вопросов документа DO-248C поясняется: «Неограниченный рекурсивный алгоритм – это алгоритм, который напрямую вызывает сам себя, то есть саморекурсия, или косвенно вызывает сам себя, то есть взаимная рекурсия, и не имеет механизма, ограничивающего число таких вызовов до завершения работы» [11]. Да-

лее в этом вопросе объясняется, что для рекурсивных алгоритмов необходимо задать верхнюю границу числа рекурсивных вызовов и показать, что доступного пространства стека достаточно для этой верхней границы [11].

7. *Осторожно используйте функции, которые могут быть вызваны повторно до завершения предыдущего вызова.* Подобно рекурсии, многие разработчики запрещают такой код. Если он допускается, что часто бывает в многопоточных программах, его применение должно напрямую трассироваться к требованиям, и такой код не должен присваивать значения глобальным переменным.
8. *Избегайте модифицирующегося кода.* Самомодифицирующийся код – это программа, которая изменяет собственный поток инструкций во время выполнения. Такой код подвержен ошибкам, его трудно читать, сопровождать и тестировать. Поэтому его следует избегать.
9. *Избегайте использования оператора goto.* Большинство стандартов кодирования для критичного по безопасности программного обеспечения запрещают goto, поскольку такой код трудно читать, трудно доказывать его корректность и он легко превращается в так называемый спагетти-код. При этом, если goto все же используется, делать это нужно редко и с большой осторожностью.[*]
10. *Обосновывайте любые случаи, когда вы решаете не следовать этим рекомендациям.* Это именно рекомендации, и иногда могут возникать ситуации, в которых оправдано нарушение одной или нескольких из них. Но следует помнить, что эти рекомендации опираются на многолетнюю практику и взаимодействие с международными сертифицирующими органами. Если принято решение им не следовать, это нужно технически обосновать и согласовать с сертифицирующим органом.

Рекомендация 16: давайте обратную связь, когда обнаруживаются проблемы в требованиях или проекте. На этапе кодирования программистам рекомендуется сообщать о любых проблемах, обнаруженных в требованиях или проекте. Этапы проектирования и кодирования тесно связаны и иногда перекрываются. На некоторых проектах проектировщик одновременно пишет код. Если проектировщики и программисты разделены, полезно включать программистов в рассмотрения требований и проекта. Это дает им возможность лучше понять требования и проект, а также заранее дать обратную связь.

После начала кодирования должен существовать организованный способ, позволяющий программистам фиксировать проблемы требований и проекта и добиваться принятия соответствующих мер. Для фиксации проблем в требованиях или проекте обычно используется процесс обработки сообщений о проблемах, но при этом нужна и упреждающая реакция на обнаруженные проблемы. Иначе возникает риск, что требования, проект и код начнут

расходиться.

Рекомендация 17: упреждающе отлаживайте код. Как будет обсуждаться в главе 9, отладка и проверка в ходе разработки, например модульное тестирование и статический анализ кода, должны выполняться уже на этапе кодирования. Не следует ждать формального тестирования, чтобы начать находить ошибки в коде.

8.2.5 Специальные темы, связанные с кодом

8.2.5.1 Стандарты кодирования

Как отмечалось в главе 5, стандарты кодирования разрабатываются на этапе планирования, чтобы определить, как именно выбранный язык программирования будет использоваться в проекте.[*] Важно разработать полные стандарты кодирования и обучить команду их применению. Обычно компании тратят на свои стандарты кодирования значительное время, потому что затем эти стандарты используются на многих проектах. Материал этой главы дает ряд понятий и вопросов, которые следует отразить в стандартах кодирования.

Я рекомендую включать в стандарты обоснование каждого правила или рекомендации, а также примеры. Программисты гораздо охотнее применяют стандарты, когда понимают, *почему* эти рекомендации вообще существуют.

Слишком часто код пишут без должного внимания к стандартам. Затем во время рассмотрений кода обнаруживаются существенные проблемы, и код приходится переделывать. Гораздо эффективнее с самого начала убедиться, что разработчики понимают стандарты и действительно их применяют.

8.2.5.2 Библиотеки, поставляемые вместе с компилятором

Большинство производителей компиляторов поставляют вместе с ними библиотеки, которые программисты используют для реализации нужных функций в коде, например математических функций. При вызове такие библиотечные функции становятся частью бортового программного обеспечения и должны удовлетворять целям DO-178C так же, как и остальное бортовое ПО. Позиция группы Certification Authorities Software Team (CAST)[*] по этой теме изложена в документе CAST-21 под названием *Compiler-Supplied Libraries*. По сути, эта позиция требует, чтобы библиотечный код соответствовал целям DO-178C: для библиотечных функций должны существовать требования, проект и тесты [14]. Обычно производители либо разрабатывают собственные библиотеки вместе со всеми поддерживающими артефактами, либо выполняют обратное проектирование библиотечного кода, поставляемого с компилятором, чтобы восстановить требования, проект и тестовые примеры. Функции, для которых нет поддерживающих артефактов, то есть требований, проекта, исходного кода, тестов и т.д., следует либо удалить из библиотеки, либо намеренно отключить. Удаление предпочтительнее: иначе такие функции могут быть случайно активированы при

следующем использовании библиотеки. Многие компании разрабатывают библиотеку целиком, чтобы затем использовать ее в нескольких проектах. Чтобы обеспечить повторное использование библиотек, разумно отделять требования, проект и тестовые данные библиотек от данных остального бортового ПО. В зависимости от проекта библиотеки может потребоваться повторно протестировать в следующем проекте, например из-за отличий в настройках компилятора или процессора. Для приложений уровня C или D иногда можно подтвердить функциональность библиотеки с помощью тестирования и опыта эксплуатации.

Подход к библиотекам целесообразно описать в Плане сертификации ПО (Plan for Software Aspects of Certification), чтобы заранее согласовать его с сертифицирующим органом.

8.2.5.3 Генераторы автокода

До этого момента в главе в основном рассматривался исходный код, написанный вручную. Если используется генератор автокода, многие вопросы из этой главы следует учитывать и при разработке самого генератора. Кроме того, в главе 13 подробнее рассматривается квалификация инструмента, которая может потребоваться, если сгенерированный код не рассматривается вручную.

8.3 Верификация исходного кода

Цели 1-6 таблицы A-5 документа DO-178C и раздел 6.3.4 посвящены верификации исходного кода. Большая часть этих целей достигается с помощью коллегиального рассмотрения кода: используется тот же базовый процесс рассмотрения, который обсуждался в главе 6, но применительно именно к коду. Ниже приведены цели таблицы A-5 DO-178C и краткое пояснение того, что по ним ожидается [1]:[*]

- *Цель 1 таблицы A-5 DO-178C:* «Исходный код соответствует требованиям низкого уровня». Для выполнения этой цели требуется сопоставить исходный код и требования низкого уровня, чтобы убедиться, что код точно реализует требования и только требования. Эта цель тесно связана с целью 5 таблицы A-5, поскольку трассируемость помогает установить такое соответствие.
- *Цель 2 таблицы A-5 DO-178C:* «Исходный код соответствует архитектуре программного обеспечения». Назначение этой цели состоит в том, чтобы убедиться, что исходный код согласован с архитектурой. Эта цель обеспечивает согласованность потоков данных и управления между архитектурой и кодом. Как будет обсуждаться в главе 9, такая согласованность важна для анализа межкомпонентной связности по данным и по управлению.
- *Цель 3 таблицы A-5 DO-178C:* «Исходный код верифицируем». Здесь речь идет о тесто-

пригодности самого кода. Код должен быть написан так, чтобы его можно было тестировать. В главе 7 перечислены характеристики тестопригодного программного обеспечения.

- *Цель 4 таблицы A-5 DO-178C: «Исходный код соответствует стандартам».* Назначение этой цели состоит в том, чтобы убедиться, что код соответствует стандартам кодирования, определенным в планах. В этой главе и в главе 5 обсуждаются стандарты кодирования. Обычно для подтверждения выполнения этой цели используют коллегиальное рассмотрение и/или средство статического анализа. Если используется средство, может потребоваться его квалификация.
- *Цель 5 таблицы A-5 DO-178C: «Исходный код трассируется к требованиям низкого уровня».* Эта цель подтверждает полноту и точность трассируемости между исходным кодом и требованиями низкого уровня. Трассируемость должна быть двунаправленной. Все требования должны быть реализованы, и не должно быть кода, который не связан трассировкой хотя бы с одним требованием. Обычно требования низкого уровня трассируются к функциям или процедурам исходного кода. Дополнительные сведения о двунаправленной трассируемости приведены в главе 6.
- *Цель 6 таблицы A-5 DO-178C: «Исходный код точен и согласован».* Это непростая цель. Ее выполнение предполагает рассмотрение кода на предмет точности и согласованности.

Однако пояснение к цели также требует верифицировать следующие аспекты: «использование стека, использование памяти, переполнение и разрядность арифметики с фиксированной точкой, арифметика с плавающей точкой, конкуренция за ресурсы и ресурсные ограничения, время выполнения в наихудшем случае, обработка исключений, использование неинициализированных переменных, управление кэшем, неиспользуемые переменные и повреждение данных из-за конфликтов задач или прерываний» [1]. Такая верификация выходит за рамки простого рассмотрения кода. В главе 9 объясняются некоторые дополнительные мероприятия по верификации, необходимые для оценки использования стека, времени выполнения в наихудшем случае, использования памяти и т.д.

8.4 Интеграция в ходе разработки

DO-178C выделяет два аспекта интеграции. Первый – интеграция в ходе разработки, то есть процессы компиляции, компоновки и загрузки. Второй – интеграция в ходе тестирования, включая интеграцию ПО с ПО и ПО с аппаратурой. Обычно интеграция начинается с объединения программных модулей внутри одной функциональной области на хост-компьютере. Затем на хост-компьютере объединяются несколько функциональных областей, а после этого программное обеспечение интегрируется на целевом оборудовании.

Эффективность интеграции доказывается в процессе тестирования. В этом разделе рассматривается интеграция в ходе разработки, см. рисунок 8.1. Интеграция на этапе тестирования рассматривается в главе 9.

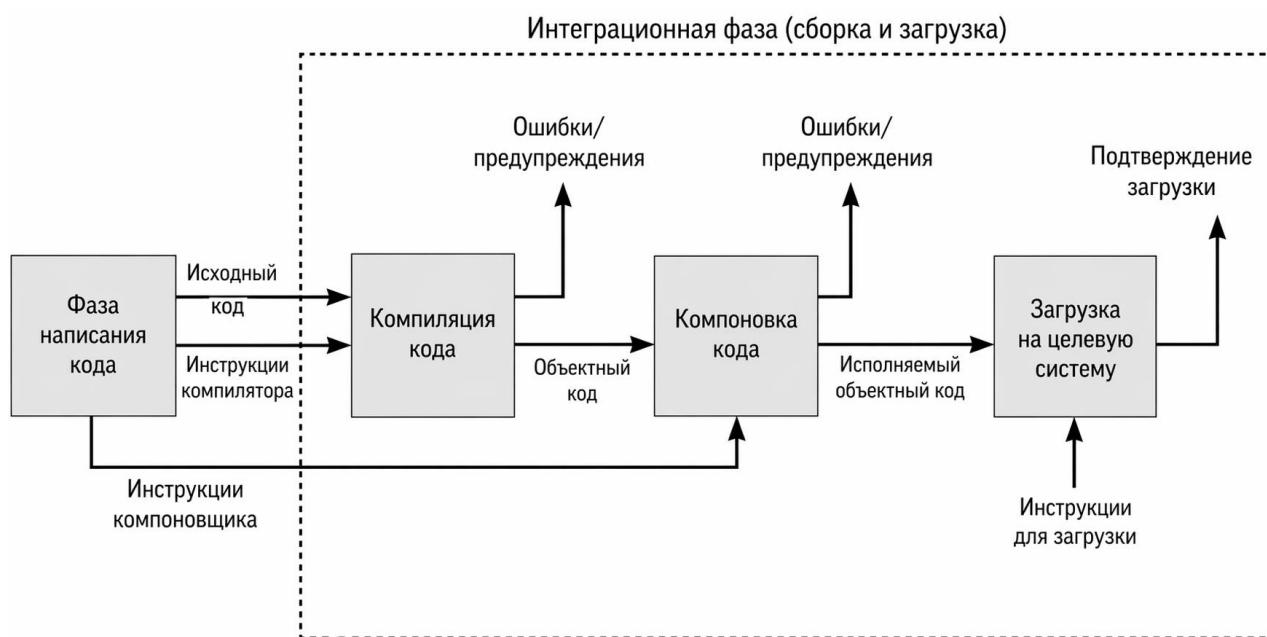


Рисунок 8.1 Этап кодирования и интеграции.

8.4.1 Процесс сборки

На рисунке 8.1 показано общее представление процесса интеграции: компиляция кода, компоновка и загрузка на целевой вычислитель. Процесс получения исполняемого объектного кода из исходного кода называется *процессом сборки*. Выходные данные этапа кодирования включают исходный код, а также инструкции по компиляции и компоновке. Инструкции по компиляции и компоновке документируются в *инструкциях по сборке*. Инструкции по сборке должны быть хорошо документированы и состоять из повторяемых шагов, поскольку они описывают процесс создания исполняемого образа, который будет использоваться при критичной по безопасности эксплуатации. Раздел 11.16.g документа DO-178C рекомендует включать инструкции по сборке в Указатель конфигурации ПО (Software Configuration Index).

Инструкции по сборке часто включают несколько сценариев, например *makefile*. Поскольку эти сценарии играют ключевую роль в создании исполняемого образа или образов, они должны находиться под управлением конфигурацией и рассматриваться на корректность так же, как и исходный код. К сожалению, рассмотрение данных компиляции и компоновки часто упускается из виду еще на этапах планирования. Некоторые организации очень тщательно работают с исходным кодом, но практически полностью пренебрегают сценариями, на которых держится процесс сборки. Объясняя термин *исходный код*, раздел 11.11 документа DO-178C говорит: «Эти данные состоят из кода, написанного на исходном языке или

языках. Исходный код используется вместе с данными компиляции, компоновки и загрузки в процессе интеграции для создания интегрированной системы или оборудования» [1]. Следовательно, данные компиляции и компоновки должны тщательно контролироваться вместе с исходным кодом. Для этого требуются верификация и управление конфигурацией соответствующих данных. Уровень программного обеспечения определяет объем верификации и управления конфигурацией.

Процесс сборки опирается на хорошо контролируемую среду разработки. Описание среды разработки должно перечислять все средства с указанием версий, аппаратные средства и настройки, включая параметры компилятора и компоновщика, используемые в среде сборки. DO-178C рекомендует документировать эту информацию в Указателе конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index). Указатель конфигурации среды ЖЦ ПО рассматривается в главе 10.

Перед сборкой программного обеспечения для выпуска большинство компаний требуют чистую сборку. В этой ситуации машину сборки *очищают*, удаляя с нее все ПО. Затем на машину загружают утвержденное ПО в соответствии с *процедурой чистой сборки*. Такая чистая сборка гарантирует, что используется утвержденная среда, а также что среду сборки можно воспроизвести заново, что важно для сопровождения. После правильной настройки машины сборки выполняются инструкции по сборке ПО и формируется версия для выпуска. Процедуры чистой сборки обычно включаются в Указатель конфигурации среды ЖЦ ПО или Указатель конфигурации ПО либо даются в этих документах по ссылке.

Один из аспектов процесса сборки, который иногда упускают, – это обработка предупреждений и ошибок компилятора и компоновщика. Инструкции по сборке должны требовать, чтобы предупреждения и ошибки анализировались после компиляции и компоновки. Инструкции по сборке также должны перечислять любые допустимые предупреждения либо описывать процесс анализа предупреждений для определения их приемлемости. Ошибки обычно неприемлемы.

По моему опыту, процедуры чистой сборки и инструкции по сборке программного обеспечения часто документированы плохо и поэтому не воспроизводятся надежно.

Процесс сборки нередко почти полностью держится на инженере, который выполняет ее едва ли не ежедневно.

Чтобы устранить этот типичный недостаток, полезно поручать выполнение процедур человеку, который их не писал и обычно не занимается сборкой. Так можно подтвердить воспроизводимость. Я рекомендую делать это как часть рассмотрения процедур сборки.

8.4.2 Процесс загрузки

Процесс загрузки определяет, как исполняемый образ или образы помещаются на целевую платформу. Обычно существуют процедуры загрузки для лаборатории, завода и воздушного судна, если программное обеспечение может загружаться в эксплуатации. Процедуры загрузки должны быть документированы и контролироваться. Раздел 11.16.k документа DO-178C поясняет, что процедуры и методы, используемые для загрузки ПО на целевое аппаратное обеспечение, должны быть документированы в Указателе конфигурации ПО.[*]

В инструкциях по загрузке должно быть указано, как проверяется полнота загрузки, как выявляются неполные загрузки, как обрабатываются поврежденные загрузки и что делать, если в ходе загрузки возникает ошибка.

Для авиационных систем многие производители используют протокол ARINC 615A [15] и высоконадежную циклическую избыточную проверку, чтобы убедиться, что программное обеспечение корректно загружено на целевую платформу.

8.5 Верификация интеграции в ходе разработки

Верификация интеграции в ходе разработки обычно включает следующие мероприятия, позволяющие убедиться в полноте и корректности процесса интеграции:

- рассмотрение данных компиляции, данных компоновки и данных загрузки, например сценариев, используемых для автоматизации сборки и загрузки;
- рассмотрение инструкций по сборке и загрузке, включая их независимое выполнение, чтобы убедиться в полноте и воспроизводимости;
- анализ данных компоновки, данных загрузки и карты памяти, чтобы убедиться в корректности аппаратных адресов, отсутствии перекрытий памяти и отсутствии пропущенных программных компонентов. Это позволяет выполнить цель 7 таблицы А-5 документа DO-178C, которая гласит: «Выходные данные процесса интеграции программного обеспечения полны и корректны» [1]. Эти анализы подробнее рассматриваются в главе 9.

Список литературы

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
2. Wikipedia, Assembly language, http://en.wikipedia.org/wiki/Assembly_language (accessed on January 2011).
3. J. Cooling, *Software Engineering for Real-Time Systems* (Harlow, U.K.: Addison-Wesley, 2003).

4. Wikipedia, Ada programming language,
 - [http://en.wikipedia.org/wiki/Ada_\(programming_language\)](http://en.wikipedia.org/wiki/Ada_(programming_language)) (accessed on January 2011).
5. Wikipedia, C programming language,
[http://en.wikipedia.org/wiki/C_\(programming_language\)](http://en.wikipedia.org/wiki/C_(programming_language)) (accessed on January 2011).
6. S. McConnell, *Code Complete*, 2nd edn. (Redmond, WA: Microsoft Press, 2004).
7. The Motor Industry Software Reliability Association (MISRA), *Guidelines for the Use of the C Language in Critical Systems*, MISRA-C:2004 (Warwickshire, U.K.: MISRA, October 2004).
8. A. Hunt and D. Thomas, *The Pragmatic Programmer* (Reading, MA: Addison-Wesley, 2000).
9. C. M. Krishna and K. G. Shin, *Real-Time Systems* (New York: McGraw-Hill, 1997).
10. C. Jones, *Software Assessments, Benchmarks, and Best Practices* (Reading, MA: Addison-Wesley, 2000).
11. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
12. G. J. Myers, *The Art of Software Testing* (New York: John Wiley & Sons, 1979).
13. C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd edn. (New York: John Wiley & Sons, 1999).
14. Certification Authorities Software Team (CAST), Compiler-supplied libraries, Position Paper CAST-21 (January 2004, Rev. 2).
15. Aeronautical Radio, Inc., Software data loader using ethernet interface, ARINC REPORT 615A-3 (Annapolis, MD: Airlines Electronic Engineering Committee, June 2007).

Рекомендуемая литература

1. S. McConnell, *Code Complete*, 2nd edn. (Redmond, WA: Microsoft Press, 2004).
2. A. Hunt and D. Thomas, *The Pragmatic Programmer* (Reading, MA: Addison-Wesley, 2000).
3. B. W. Kernighan and R. Pike, *The Practice of Programming* (Reading, MA: Addison-Wesley, 1999).

4. The Motor Industry Software Reliability Association (MISRA), *Guidelines for the Use of the C Language in Critical Systems*, MISRA-C:2004 (Warwickshire, U.K.: MISRA, October 2004).

5. ISO/IEC PDTR 15942, *Guide for the Use of the Ada Programming Language in High Integrity Systems*,

<http://anubis.dkuug.dk/JTC1/SC22/WG9/documents.htm> (July 1999).

6. L. Hattan, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems* (Maidenhead, U.K.: McGraw-Hill, 1995).

7. S. Maguire, *Writing Solid Code* (Redmond, WA: Microsoft Press, 1993).

*Он известен как справочное руководство по языку Ada (*Ada Reference Manual*, ARM), то есть справочное руководство по языку (*Language Reference Manual*, LRM).

*Хотя книга Стива Макконнелла *Code Complete* [6] не посвящена специально критичному по безопасности программному обеспечению, в ней приведены подробные рекомендации по каждой из этих тем.

†В книге Стива Макконнелла *Code Complete* [6] приведены несколько советов, как избегать ошибок при работе с указателями.

*В своей книге *Code Complete* [6] Стив Макконнелл дает несколько рекомендаций по осмот- рительному использованию goto.

*Раздел 11.8 документа DO-178C объясняет ожидаемое содержимое стандартов.

*Это международная группа сертифицирующих органов, стремящаяся согласовать свои по- зиции по авиационному программному обеспечению и авиационной электронной аппара- туре в своих документах.

*Цель 7 таблицы A-5 документа DO-178C объясняется в главе 9.

*Этого не было в DO-178B.

Глава 9. Верификация программного обеспечения

Сокращения

Сокращение	Расшифровка
BVA	Анализ граничных значений
CAST	Группа сертифицирующих органов по программному обеспечению (Certification Authorities Software Team)
KK1	Категория контроля 1 (control category 1)
KK2	Категория контроля 2 (control category 2)
CCB	Совет по управлению изменениями (change control board)
DC/CC	Связность по данным и управлению между компонентами (data coupling and control coupling)
DP	Дискуссионный документ (discussion paper)
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
IMA	Интегрированная модульная авионика (Integrated Modular Avionics)
ISR	Подпрограмма обслуживания прерывания (interrupt service routine)
MC/DC	Модифицированное покрытие условий/решений (modified condition/decision coverage)
OOT	Объектно-ориентированная технология (object-oriented technology)
PR	Сообщение о проблеме (problem report)
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
RAM	Оперативная память (random access memory)
SAS	Итоговый отчёт разработки ПО (Software Accomplishment Summary)
SCMP	План управления конфигурацией ПО (Software Configuration Management Plan)
SDP	План разработки ПО (Software Development Plan)
SECI	Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index)

Сокращение	Расшифровка
SQA	Гарантия качества ПО (software quality assurance)
SQAP	План гарантии качества ПО (Software Quality Assurance Plan)
SVCP	Тестовые примеры и процедуры верификации ПО (Software Verification Cases and Procedures)
SVP	План верификации ПО (Software Verification Plan)
SVR	Результаты верификации ПО (Software Verification Results)
TRR	Рассмотрение готовности к тестированию (test readiness review)
WCET	Наихудшее время выполнения (worst-case execution timing)

9.1 Введение

Верификация – неотъемлемый процесс, который применяется на протяжении всего жизненного цикла программного обеспечения. Она начинается на этапе планирования и продолжается вплоть до выпуска в производство, а затем и на этапе сопровождения.

Глоссарий DO-178C (КТ-178С) определяет верификацию как «оценку выходных данных процесса для подтверждения их корректности и соответствия входным данным и стандартам, определенным для этого процесса» [1]. Указания DO-178C по верификации объединяют рассмотрения, анализы и тестирование. Рассмотрения и анализы оценивают корректность, полноту и проверяемость результатов на каждом этапе жизненного цикла: при планировании, работе с требованиями, проектировании, кодировании и интеграции, разработке тестов и выполнении тестирования. В общем случае рассмотрение дает качественную оценку корректности, а анализ дает воспроизводимое свидетельство корректности [1]. Тестирование – это «процесс выполнения заданий системой или ее компонентом для проверки соответствия заданным требованиям и для выявления ошибок» [1]. Все три подхода широко применяются при верификации критического по безопасности программного обеспечения. При верификации обычно используют термины *ошибка* (error), *неисправность* (fault) и *отказ* (failure). Глоссарий DO-178C определяет их следующим образом [1]:

- «Error (ошибка) – применительно к программному обеспечению это ошибка в требованиях, проекте или коде».
- «Fault (неисправность) – проявление ошибки в программном обеспечении. Неисправность, если она возникает, может явиться причиной отказа».
- «Failure (отказ) – неспособность системы или ее компонента выполнять требуемую функцию в пределах заданных ограничений. Отказ может быть результатом проявления неисправности».

Главная задача верификации – выявить ошибки, чтобы их можно было исправить до того, как они приведут к неисправностям или отказам. Чтобы находить ошибки как можно раньше, верификацию нужно начинать уже на ранних стадиях жизненного цикла программного обеспечения.

9.2 Важность верификации

Рональду Рейгану приписывают, или по крайней мере он сделал широко известной, фразу: «Доверяй, но проверяй». Джордж Романски формулирует это так: «Многие люди могут писать программное обеспечение, но не многие доверили бы ему свою жизнь, пока оно не прошло верификацию». Верификация важна в любом жизненном цикле ПО, но для критического по безопасности ПО она особенно важна. В DO-178C более половины целей классифицируются как цели верификации. Чем критичнее ПО, тем больше требуется мероприятий по верификации и тем выше должна быть уверенность в том, что ошибки выявлены и устранены.

По словам многих руководителей проектов, с которыми я общаюсь и работаю, для критического по безопасности программного обеспечения более половины бюджета проекта уходит на верификацию. К сожалению, во многих проектах верификацию воспринимают как необходимое зло: на нее выделяют слишком мало людей и относятся к ней как к формальности для проставления галочки. Несмотря на массу свидетельств того, что раннее обнаружение ошибок экономит время и деньги, многие проекты по-прежнему стараются просто *проскочить* рассмотрения, чтобы удовлетворить сертифицирующий орган, а тестирование откладывают на самый конец. Слишком часто рассмотрения, анализы и тестирование отдаются младшим инженерам, чтобы они *просто закрыли задачу*. У меня нет ничего против младших инженеров; более того, когда-то я и сама была таким инженером. Однако хорошие навыки верификации появляются не сразу, а со временем и при должном обучении. В верификацию должны быть вовлечены опытные инженеры. Использовать менее опытных инженеров тоже нормально, но именно те, кто уже прошел через несколько проектов, обычно находят ошибки, которые, если их вовремя не заметить, способны серьезно испортить проект.

Коллега рассказал мне, как однажды зашел в лабораторию и увидел младшего инженера, выполнявшего тестирование небольшой функции. Хотя тестируемый небольшой фрагмент системы работал правильно, инженер не заметил предупреждений в кабине, которые отображались из-за серьезного отказа в системе. Это был типичный случай, когда неопытный инженер получил свою *галочку* за порученную задачу, но упустил из виду базовое понимание системы. Это похоже на ситуацию, когда механик проверяет автомобиль и говорит, что шины накачаны правильно, в то время как из двигателя валит дым. Сложность в том, что-

бы видеть систему целиком и не попадать в ловушку туннельного зрения, когда внимание застревает на одной цели.

С точки зрения безопасности верификация абсолютно необходима. Она используется для выполнения нормативных требований и подтверждает, что программное обеспечение выполняет назначенную функцию – и только ее. Проще говоря, без качественной верификации гарантия разработки не имеет особой ценности, потому что именно верификация формирует доверие к продукту.

9.3 Независимость и верификация

Прежде чем переходить к деталям верификации, кратко рассмотрим тему независимости. Если вам когда-либо приходилось редактировать собственную работу, вы знаете, как трудно находить в ней коварные опечатки. Вы знаете, как текст *должен* выглядеть, и поэтому просто их пропускаете. То же самое справедливо и при верификации программных данных. Поэтому по мере роста критичности программного обеспечения растет и требуемая независимость между мероприятиями.

DO-178C определяет независимость следующим образом:

Разделение ответственности, обеспечивающее объективную оценку. (1) Для мероприятий процесса верификации программного обеспечения независимость достигается тогда, когда верификационное мероприятие выполняется лицом или лицами, не участвовавшими в разработке верифицируемой единицы, а также инструментом или инструментами, обеспечивающими эквивалентные человеку верификационные действия. (2) Для процесса гарантии качества ПО независимость также включает полномочия для обеспечения выполнения корректирующих действий [1].

Эта глава касается первой половины определения, то есть *независимости верификации*. Как видно из определения, независимость верификации не требует отдельной организации, а требует лишь отдельного лица, лиц или инструмента. В главе 11 будет рассмотрена вторая половина определения, относящаяся к *независимости гарантии качества ПО*.

В таблицах А-3 – А-7 DO-178C цели верификации, требующие независимости, отмечены закрашенным кружком. Для уровня А независимость требуется для 25 целей верификации; для уровня В независимо должны выполняться только 13 целей верификации; для уровней С и D независимость не требуется ни для одной цели верификации. Для таблиц А-3 – А-5 DO-178C независимость обычно обеспечивается тем, что данные рассматривает не тот, кто их писал. Однако цели 6 и 7 таблицы А-5 DO-178C обычно требуют также некоторого анализа и/или тестирования. Две независимые цели из таблицы А-6 DO-178C обычно выполняются за счет того, что тесты пишет не тот человек, который писал код. Для таблицы

A-7 DO-178C все цели уровня А и три цели уровня В требуют независимости. Для таблицы A-7 независимость обычно обеспечивается сочетанием рассмотрений, для целей 1-4, и анализов, для целей 5-9.[*]

Дискуссионный документ DO-248C #19 поясняет типичную интерпретацию независимости верификации. Этот документ был основан на документе CAST-26 группы Certification Authorities Software Team (CAST)[†], поэтому он дает представление о том, как сертифицирующие органы обычно трактуют независимость верификации [2]. Следует отметить, что документ CAST требовал большей независимости между действиями разработки, чем это указано в DO-248C DP #19. DO-248C разъясняет, что независимость разработки требуется только между разработчиком исходного кода и спецификациями тестов; другие действия разработки не требуют независимого лица или инструмента [3]. DO-178C также проясняет это в разделе 6.2.е, где сказано: «Для независимости лицо, создавшее набор тестовых примеров на основе требований низкого уровня, не должно быть тем же лицом, которое разработало соответствующий исходный код по этим требованиям низкого уровня» [1].

Как уже отмечалось выше, качество верификации ограничено качеством человека или инструмента, который ее выполняет. Поэтому важно использовать квалифицированный персонал или эффективные инструменты. В некоторых случаях инструменты могут требовать квалификации, см. главу 13.

В DO-178C есть цели, для которых независимость не требуется; однако даже в этих случаях ее стоит обеспечивать, особенно для программного обеспечения уровней А и В. Опыт показывает, что независимая верификация является одним из самых эффективных способов рано находить ошибки. Многие зрелые компании применяют независимость при рассмотрении требований и проекта на всех уровнях, даже когда это не требуется, потому что это эффективно выявляет ошибки и в долгосрочной перспективе экономит время и деньги. Кроме того, как будет обсуждаться далее, у верификаторов обычно иной образ мышления, чем у разработчиков, поэтому они нередко находят больше ошибок, чем разработчик при рассмотрении собственной работы.

9.4 Рассмотрения

Теперь рассмотрим три подхода к верификации в DO-178C. В этом разделе речь пойдет о рассмотрениях; в следующих двух разделах – об анализах и тестировании.

Как отмечалось ранее, рассмотрение – это качественная оценка артефакта на предмет корректности и соответствия требуемым целям [1]. В главах 6-8 этот подход применялся к требованиям, проекту и коду. Большинство компаний используют коллегиальные рассмотрения для проверки своих артефактов. В таком рассмотрении участвует группа рецензентов, и у каждого есть своя конкретная цель и область внимания. В главе 6 были даны рекомендации

по проведению эффективного коллегиального рассмотрения. Как правило, рассмотрение опирается на применимые стандарты и включает контрольный перечень, направляющий работу рецензентов. Рецензенты документируют свои замечания, после чего каждое замечание надлежащим образом обрабатывается, закрывается и проверяется. Если по итогам рассмотрения вносятся существенные изменения, может потребоваться полностью провести новое рассмотрение данных.

9.4.1 Рассмотрение планирования программного обеспечения

В главе 5 рассматривалась разработка пяти планов по программному обеспечению и трех стандартов. Для уровней А, В и С цели 6 и 7 таблицы А-1 DO-178С требуют, чтобы планы соответствовали DO-178С и были согласованы между собой [1]. Обычно эти цели достигаются через рассмотрение планов и стандартов, чтобы убедиться в их согласованности и соответствии указаниям DO-178С. Предпочтительной практикой является сначала рассмотрение каждого документа по отдельности, а затем рассмотрение всех планов и стандартов вместе, когда они уже написаны. Без совместного рассмотрения документов можно не заметить несогласованности и пробелы. Я часто подчеркиваю: «Не позволяйте сертифицирующему органу или уполномоченному представителю стать первым, кто прочитает ваши планы и стандарты в совокупности».

9.4.2 Рассмотрения требований, проекта и кода

В главах 6-8 рассматривались характеристики хороших требований, проекта и кода. Кроме того, в этих главах были указаны цели из таблиц А-3 – А-5 DO-178С для верификации каждого из артефактов. Рассмотрение каждого элемента данных жизненного цикла должно проводиться как можно раньше, чтобы ошибки выявлялись и устранялись заранее. Требования, проект и код часто неформально рассматриваются несколько раз до формального коллегиального рассмотрения. Кроме того, разработчики обычно выполняют собственное отладочное тестирование до коллегиального рассмотрения кода. Неформальные рассмотрения и отладочные мероприятия позволяют раньше отсеять крупные и наиболее очевидные ошибки, сокращая объем поздних переделок.

9.4.3 Рассмотрения тестовых данных

Следует отметить, что коллегиальные рассмотрения также используются для верификации тестовых примеров и процедур, процедур и результатов анализа, а также результатов тестирования. Эти рассмотрения описаны далее в этой главе.

9.4.4 Рассмотрение других элементов данных

Рассмотрения также используются, чтобы убедиться в точности и корректности других важных элементов данных, например указателя конфигурации среды ЖЦ ПО, указателя конфигурации ПО и итогового отчета разработки ПО.

9.5 Анализы

Анализ – это действие по верификации, дающее воспроизводимое свидетельство корректности [1]. На протяжении жизненного цикла критического по безопасности программного обеспечения выполняется несколько видов анализа. Для соответствия DO-178C необходимы две основные категории анализа: (1) анализы кода и интеграции и (2) анализы покрытия. В зависимости от выбранного подхода к верификации часто выполняются и другие анализы. Типичные анализы кода и интеграции описаны в этом разделе. Анализ покрытия рассматривается далее в этой главе, см. раздел 9.7.

Инженеры, и я в том числе, иногда слишком свободно употребляют слова *анализ* и *анализировать*. Однако для соответствия DO-178C анализ имеет вполне конкретный смысл: он должен быть *воспроизводимым*, а значит, его нужно хорошо документировать. При рассмотрении данных как уполномоченный представитель FAA я часто нахожу существенные проблемы в анализах. Нередко так называемый анализ вообще не записан; следовательно, он не является воспроизводимым. Точно так же, когда анализы *записаны*, в них часто отсутствуют критерии успеха. Анализы в стиле *дым и зеркала*, *размахивание руками* или *черная магия* неприемлемы. Анализ должен иметь процедуры и результаты. Процедуры включают следующее [4]:

- назначение, критерии и связанные требования;
- подробные инструкции по выполнению анализа;
- критерии приемлемости и завершения анализа.

Результаты анализа включают следующее [4]:

- идентификацию процедуры анализа;
- идентификацию анализируемого элемента данных;
- указание того, кто выполнил анализ;
- результаты анализа и подтверждающие данные;
- корректирующие действия, сформированные по результатам анализа;
- заключение анализа с обосновывающими данными.

Анализы должны выполняться с надлежащим уровнем независимости, как это указано в таблицах приложения A DO-178C. Если анализ используется вместо тестирования, то требуется тот же уровень независимости, что и для соответствующей цели тестирования.

Анализы следует начинать настолько рано, насколько это практически возможно, чтобы выявлять проблемы как можно раньше. Обычно, как только устанавливается базовая версия кода, можно начинать аналитические действия. Окончательные анализы выполняются только после финализации программного обеспечения, но предварительные анализы уже могут

выявить часть проблем. Недавно я консультировала проект, на котором выяснилось, что их наихудшее время выполнения (worst-case execution timing, WCET) составляло примерно 110%-130% от требуемого; по сути, у них были отрицательные запасы, а это означало, что функции могли не успевать выполняться. К сожалению, проблему обнаружили за несколько недель до целевой даты сертификации, и это привело к серьезным задержкам. Когда речь идет об анализах, идеальная картина и реальность часто расходятся, обычно из-за нехватки персонала и сжатых сроков; однако чем дольше команда откладывает начало анализов, тем больший риск она принимает на себя.

Ниже кратко описаны типичные интеграционные анализы, выполняемые для соответствия DO-178C.[*] Результаты всех интеграционных анализов обычно обобщаются в результатах верификации ПО, которые поясняются в разделе 9.6.7. Некоторые анализы также обобщаются в итоговом отчете разработки ПО как характеристики программного обеспечения. Анализы, которые включаются в итоговый отчет разработки ПО, будут отдельно отмечены далее. Сам итоговый отчет разработки ПО рассматривается в главе 12.

9.5.1 Анализ наихудшего времени выполнения

Знание временных характеристик программы необходимо для успешного проектирования и работы систем реального времени. Критически важная временная характеристика – наихудшее время выполнения (worst-case execution timing, WCET). Это максимально возможное время, которое может потребоваться для завершения выполнения набора задач на заданном процессоре в целевой среде. Анализ наихудшего времени выполнения проводят, чтобы подтвердить, что это время укладывается в выделенный бюджет. Хотя конкретный подход зависит от архитектуры программного обеспечения и системы, на практике это время обычно и анализируют, и измеряют.

При анализе наихудшего времени выполнения обычно рассматривают каждую ветвь и каждый цикл в коде, чтобы определить путь или пути выполнения с наихудшим временем прохождения через код. Затем время для каждой ветви и каждого цикла на этом пути суммируется, что и дает наихудшее время выполнения.[*] Это время проверяется относительно временных ограничений, выделенных в требованиях. Анализ необходимо подтверждать реальными измерениями времени. Есть несколько факторов, усложняющих такой анализ, включая прерывания, алгоритмы с несколькими шагами принятия решений, использование кэша данных или команд, методы планирования и применение операционной системы реального времени [3]. Эти факторы следует учитывать при разработке программного обеспечения, чтобы гарантировать, что наихудшее время выполнения останется в требуемых пределах. Использование кэш-памяти, например кэшей L1, L2 и L3, или конвейерной обработки дополнительно усложняет такой анализ и требует дополнительной проверки, чтобы влияние

кэша и конвейера на время выполнения было понятным.[†]

Нередко для поиска пути с наихудшим временем выполнения используют инструменты. В таком случае необходимо определить точность инструмента. Существует несколько подходов к подтверждению точности инструмента, включая: (1) ручную проверку результата работы инструмента, (2) квалификацию инструмента или (3) параллельный запуск независимого инструмента и сравнение результатов. Следует помнить, что при использовании кэша или конвейерной обработки инструменты сталкиваются с теми же трудностями, что и ручной анализ.

Подход к анализу наихудшего времени выполнения и его результаты обобщаются в итоговом отчете разработки ПО как часть *характеристик программного обеспечения*, как будет обсуждаться в главе 12.

9.5.2 Анализ запасов памяти

Анализ запасов памяти выполняют, чтобы подтвердить, что для штатной эксплуатации изделия и будущего роста есть достаточный резерв. Анализируется вся используемая память, включая энергонезависимую память (non-volatile memory, NVM), оперативную память (random access memory, RAM), кучу, стек и любую динамически выделяемую память, если она используется.

Подход к анализу запасов памяти и его результаты обобщаются в итоговом отчете разработки ПО как часть *характеристик программного обеспечения*.

Например, использование стека обычно анализируют по исходному коду: определяют дерево вызовов функций с максимальной глубиной как при штатной обработке, так и при обработке прерываний. Затем по этим функциям определяют максимальный объем стековой памяти для объединенных деревьев вызовов. Если это делается на уровне исходного кода, то требуется дополнительный анализ, чтобы определить, сколько дополнительных данных используется в каждой функции для сохраненных регистров, формальных параметров, локальных переменных, данных адреса возврата и любых промежуточных результатов, необходимых компилятору. Анализ стека по исполняемому образу учитывает это автоматически. Затем рассчитанный объем использования стека сравнивают с доступным объемом стека, чтобы подтвердить наличие достаточного запаса. Для секционированного или многозадачного программного обеспечения этот анализ повторяют для всех разделов и задач, поскольку у каждого из них будет свой стек. Как и в случае с временным анализом, анализ использования стека обычно подтверждается реальными измерениями, если только он не выполняется квалифицированным инструментом анализа стека.

9.5.3 Анализ компоновки и карты памяти

Анализ компоновки подтверждает, что модули программной сборки правильно размещены в сегментах памяти процессора, заданных соответствующими командными файлами компоновщика. Анализ компоновки обычно включает проверку файла карты памяти, чтобы убедиться в следующем:

- начало размещения, максимальная длина и атрибуты каждого сегмента, выделенного компоновщиком, соответствуют определению сегмента, заданному в командном файле компоновщика;
- ни один из выделенных сегментов не перекрывается с другим;
- фактическая суммарная длина каждого выделенного сегмента меньше или равна максимальной длине, заданной в командном файле компоновщика;
- различные секции, определенные каждым модулем исходного кода, корректно размещаются компоновщиком в соответствующих сегментах;
- присутствуют только ожидаемые объектные модули, полученные из исходного кода;
- присутствуют только ожидаемые процедуры, таблицы и переменные из скомпонованных объектных модулей.

Компоновщик присваивает корректные значения выходным символам компоновки.

В системе интегрированной модульной авионики (Integrated Modular Avionics, IMA) часть этих проверок может также выполняться относительно конфигурационных данных.[*]

9.5.4 Анализ загрузки

Анализ загрузки иногда выполняют вместе с анализом компоновки. Он подтверждает следующее:

- все программные компоненты собраны и загружены в правильное место. Недопустимое программное обеспечение не загружается на целевую систему;
- некорректное или поврежденное программное обеспечение не будет выполняться. Данные загрузки корректны.

9.5.5 Анализ прерываний

Для систем реального времени часто выполняют анализ прерываний, чтобы подтвердить два свойства: (1) все прерывания, разрешенные программным обеспечением, корректно обрабатываются соответствующими подпрограммами обслуживания прерываний (interrupt service routine, ISR), определенными в ПО; (2) неиспользуемых подпрограмм обслуживания прерываний нет. Анализ прерываний включает изучение исходного кода, определение набора разрешенных прерываний и подтверждение того, что для каждого разрешенного преры-

вания существует соответствующая подпрограмма. Каждую такую подпрограмму обычно анализируют, чтобы подтвердить следующее:

- подпрограмма обслуживания прерывания корректно расположена в памяти;
- в начале подпрограммы обслуживания прерывания сохраняется контекст системы;
- операции, выполняемые внутри подпрограммы обслуживания прерывания, соответствуют данному физическому прерыванию; например, в контексте прерывания не допускаются блокирующие операции;
- все критичные по времени операции внутри подпрограммы обслуживания прерывания завершаются до того, как может возникнуть любое другое прерывание;
- перед передачей данных в контуры управления прерывания запрещаются. Время, в течение которого прерывания запрещены ради передачи данных, минимизируется. В конце подпрограммы обслуживания прерывания контекст системы восстанавливается.

9.5.6 Математический анализ

Хотя это требуется не всегда, некоторые команды выполняют *математический анализ*, чтобы убедиться, что вычисления не влияют отрицательно на работу программного обеспечения. Иногда математический анализ выполняют как часть рассмотрения кода, поэтому его не всегда оформляют как отдельный анализ. В таком случае это следует отметить в контрольном перечне рассмотрения кода. Обычно анализ включает рассмотрение каждой арифметической или логической операции в коде, чтобы:

- определить переменные, входящие в каждую арифметическую или логическую операцию. Для каждой переменной проанализировать объявление, включая масштабирование, и убедиться, что переменные, используемые в операции, объявлены правильно и масштабированы корректно, с достаточным разрешением;
- подтвердить, что переполнение арифметической операции не может произойти.

В некоторых проектах для выполнения или дополнения такого анализа используют статические анализаторы кода. Кроме того, если используются математические библиотеки, их тоже потребуется верифицировать на корректность, см. раздел 8.2.5.2. Особый интерес представляет поведение операций с плавающей точкой на границах диапазона. Типичный современный процессор может поддерживать такие состояния чисел с плавающей точкой, как положительный и отрицательный ноль, положительная и отрицательная бесконечность, а также денормализованные числа и значения *NaN* (not a number, не число).[*] Поведение алгоритмов с плавающей точкой, которые могут получить такие значения, должно быть задано в требованиях к робастности и верифицировано.

9.5.7 Анализ ошибок и предупреждений

Как отмечалось в главе 8, во время сборки компилятор и компоновщик могут выдавать ошибки и предупреждения. Ошибки необходимо устранять, однако некоторые предупреждения могут быть допустимыми. Любые неустраненные предупреждения, не обоснованные в процедурах сборки, нужно проанализировать и подтвердить, что они не влияют на предусмотренную функциональность программного обеспечения.

9.5.8 Анализ обособления

Если система включает обособление, часто выполняется анализ обособления, чтобы подтвердить робастность этого обособления. Этот анализ особенно актуален для систем интегрированной модульной авионики (ИМА) и обособленных операционных систем. Обсуждение анализа обособления приведено в главе 21.

9.6 Тестирование программного обеспечения

Тестирование по требованиям высокого и низкого уровней является одним из основных мероприятий, обязательных для соответствия DO-178C.[†] Таблица A-6 DO-178C суммирует цели тестирования. Эти цели включают разработку и выполнение тестовых примеров и процедур для подтверждения следующего [1]:

- *цель 1 таблицы A-6 DO-178C:* «Исполняемый объектный код соответствует требованиям высокого уровня».
- *цель 2 таблицы A-6 DO-178C:* «Исполняемый объектный код робастен по отношению к требованиям высокого уровня».
- *цель 3 таблицы A-6 DO-178C:* «Исполняемый объектный код соответствует требованиям низкого уровня».
- *цель 4 таблицы A-6 DO-178C:* «Исполняемый объектный код робастен по отношению к требованиям низкого уровня».
- *цель 5 таблицы A-6 DO-178C:* «Исполняемый объектный код совместим с целевым вычислителем».

Тестирование – лишь часть общих мероприятий по верификации, но часть важная и очень трудоемкая, особенно для более высоких уровней программного обеспечения. Поэтому этой теме посвящены четыре раздела:

- в разделе 9.6 подробно рассматриваются темы тестирования, включая: (1) назначение тестирования программного обеспечения, (2) обзор указаний DO-178C по тестированию, (3) обзор стратегий тестирования, (4) планирование тестирования, (5) разработку тестов, (6) выполнение тестирования, (7) отчетность по тестированию, (8) трассирова-

мость тестирования, (9) регрессионное тестирование, (10) пригодность к тестированию и (11) автоматизацию тестирования;

- в *разделе 9.7* объясняется верификация самих работ по тестированию, включая: (1) рассмотрение тестовых примеров и процедур, (2) рассмотрение результатов тестирования, (3) анализ покрытия требований и (4) анализ структурного покрытия. В таблице А-7 DO-178С это называется *verification of verification*, то есть верификацией верификации;
- *раздел 9.8* содержит предложения по регистрации сообщений о проблемах, поскольку тестирование как раз и предназначено для их выявления;
- *раздел 9.9* завершает главу рекомендациями по общему комплексу работ по тестированию.

9.6.1 Назначение тестирования программного обеспечения

Назначение тестирования программного обеспечения – выявлять ошибки, допущенные на этапах разработки. Хотя некоторым такой взгляд кажется излишне негативным, опыт показывает, что *тестирование, ориентированное на успех*, неэффективно. Если человек стремится доказать, что ПО работает правильно, он, скорее всего, это докажет. Но качеству изделия в целом это помогает мало. Как я часто говорю, люди формата «Да, сэр!» обычно не становятся хорошими тестировщиками.

Многие считают *успешным* такой тест, который не обнаружил ошибок. Однако с точки зрения тестирования верно обратное: успешный тест – это тест, который находит ошибки. Обычно тестирование – единственная часть проекта, не сосредоточенная напрямую на успехе. Тестировщики, наоборот, пытаются сломать или разрушить программное обеспечение. Кто-то считает такой настрой циничным и пытается придать тестированию более позитивный оттенок. Но чтобы делать эту работу правильно, нужно удерживать внимание на поиске отказов. Если ошибки не найдены, их нельзя исправить, и они уйдут в эксплуатацию [6,7].

По своей природе тестирование является разрушительным процессом. Хорошие тестировщики охотятся за ошибками и используют *творческое разрушение*, чтобы ломать программное обеспечение, а не демонстрировать его корректность. Эдвард Кит пишет:

Тестирование – это позитивное и творческое усилие по разрушению. Нужны воображение, настойчивость и сильное чувство миссии, чтобы систематически находить слабые места в сложной структуре и демонстрировать ее отказы. Именно поэтому особенно трудно тестировать собственную работу. Вполне естественно, что мы не хотим находить ошибки в собственном материале [8].

Не каждый человек эффективен в тестировании. Более того, по моему опыту, хорошие те-

стировщики составляют меньшинство. Большинство людей хотят заставить объекты работать, а не разбирать их на части [7]. Однако Алан Пейдж и соавторы отмечают, что у эффективных тестировщиков другая ДНК, чем у большинства разработчиков. В ДНК тестировщика входят: (1) естественная способность мыслить на уровне системы, (2) навыки декомпозиции проблем, (3) страсть к качеству и (4) любовь к тому, чтобы понять, как что-то работает и как это сломать [9]. Хотя тестировщиков часто воспринимают как пессимистов, их миссия критически важна. Они ищут изъяны в программном обеспечении, чтобы проблемы можно было исправить и поставить качественное и безопасное изделие.

DO-178C делает упор на тестировании, основанном на требованиях, чтобы подтвердить выполнение требований и отсутствие чего-либо сверх требований. Однако важно помнить, что требования описывают, как программа должна себя вести, когда она реализована без ошибок. «Они не говорят нам, каких ошибок следует ожидать и как проектировать тесты для их обнаружения» [6]. Хорошие тестировщики предвосхищают ошибки и пишут тесты, чтобы их находить. «Ошибки программного обеспечения – это человеческие ошибки. Поскольку любая человеческая работа, особенно сложная, связана с ошибками, тестирование учитывает этот факт и сосредоточивается на выявлении ошибок наиболее продуктивным и эффективным способом, какой только можно придумать» [8]. Чем жестче тестирование, тем больше уверенности можно иметь в качестве изделия.

Некоторые полагают, что программное обеспечение можно *полностью протестировать*. Но это заблуждение. *Полное тестирование* означало бы, что проверен каждый аспект ПО, отработан каждый сценарий и обнаружена каждая неисправность. Однако в книге *Testing Computer Software* Сэм Канер и соавторы отмечают, что полностью протестировать ПО невозможно по трем причинам [10]:

- область возможных входных данных слишком велика, чтобы проверить ее полностью;
- возможных путей прохождения по программе слишком много, чтобы проверить их все;
- вопросы пользовательского интерфейса, а значит и вопросы проекта, слишком сложны, чтобы протестировать их полностью.

Книга Кита *Software Testing in the Real World* содержит несколько аксиом тестирования; ниже они приведены в кратком виде [8]:

- тестирование используется для демонстрации наличия ошибок, но не их отсутствия;
- один из самых трудных аспектов тестирования – понять, когда следует остановиться. Тестовые примеры должны включать... определение ожидаемого выхода или результата. Тестовые примеры необходимо писать как для недопустимых и неожиданных, так и для допустимых и ожидаемых входных условий;

- тестовые примеры должны разрабатываться так, чтобы формировать желаемые выходные условия, а не только входы. Иными словами, в ходе тестирования должно охватываться пространство выходов, а не только пространство входов;
- чтобы находить максимальное число ошибок, тестирование должно быть независимым мероприятием.

9.6.2 Обзор указаний DO-178C по тестированию программного обеспечения

Теперь, когда общее назначение тестирования программного обеспечения рассмотрено, посмотрим, что говорит DO-178C. DO-178C делает акцент на *тестировании, основанном на требованиях*: тесты разрабатываются и выполняются, чтобы показать, что требования выполняются, и чтобы убедиться в отсутствии непредусмотренной функциональности. Чем критичнее ПО, тем строже должна быть программа тестирования.

9.6.2.1 Методы тестирования, основанного на требованиях

Раздел 6.4.3 DO-178C предлагает три метода тестирования, основанного на требованиях.

1. *Тестирование интеграции ПО с аппаратурой, основанное на требованиях* [1]: в этом методе тесты выполняются на целевом вычислителе, чтобы выявить ошибки, проявляющиеся при работе программного обеспечения в своей рабочей среде. Многие ошибки можно обнаружить только в целевой среде. Среди функциональных областей, проверяемых при таком тестировании, обычно находятся обработка прерываний, временные характеристики, реакция на переходные процессы или отказы аппаратуры, проблемы шины данных или иные конфликты за ресурсы, встроенный контроль, интерфейсы аппаратуры и ПО, поведение контуров управления, аппаратные устройства, управляемые ПО, отсутствие переполнения стека, механизмы полевой загрузки и обособление ПО. Обычно этот метод реализуется запуском тестов по требованиям высокого уровня с нормальными входами и аномальными входами для проверки робастности на целевом вычислителе.
2. *Тестирование интеграции ПО, основанное на требованиях* [1]: этот метод сосредоточен на взаимосвязях внутри программного обеспечения и должен подтвердить, что программные *компоненты*, обычно функции, процедуры или модули, корректно взаимодействуют и удовлетворяют требованиям и архитектуре. Такое тестирование ищет ошибки процесса интеграции и интерфейсов компонентов, например повреждение данных, ошибки инициализации переменных и констант, ошибки последовательности событий или операций и так далее. Обычно это тестирование строится на основе тестирования интеграции ПО с аппаратурой по требованиям высокого уровня. Однако для проработки архитектуры ПО может потребоваться добавить дополнительные тесты, основанные на требованиях; аналогично, для дополнения тестирования интеграции ПО с

аппаратурой может потребоваться и тестирование более низкого уровня.

3. *Тестирование по требованиям низкого уровня* [1]: этот метод обычно сосредоточен на соответствии требованиям низкого уровня. Он рассматривает функциональность низкого уровня, такую как соответствие алгоритмов и их точность, работу циклов, корректность логики, комбинации входных условий, правильную реакцию на поврежденные или отсутствующие входные данные, обработку исключений, последовательности вычислений и тому подобное.

9.6.2.2 Нормальные тесты и тесты на робастность

DO-178C также предусматривает разработку двух видов тестовых примеров: нормальных тестовых примеров и тестовых примеров на робастность.[*]

9.6.2.2.1 Нормальные тестовые примеры

Нормальные тестовые примеры ищут ошибки в программном обеспечении при нормальных, ожидаемых условиях и входах. DO-178C требует нормальных тестовых примеров по требованиям высокого и низкого уровней для уровней А, В и С, а также по требованиям высокого уровня для уровня D. Нормальные тестовые примеры пишутся по требованиям, чтобы проверить допустимые значения переменных, с использованием разбиения на классы эквивалентности и анализа граничных значений, которые рассматриваются в разделах 9.6.3.1, 9.6.3.2, проверить работу временных функций в нормальных условиях, отработать переходы состояний при штатной работе и подтвердить корректность использования переменных в нормальном диапазоне и булевых операторов, когда требования выражены логическими уравнениями [1].

9.6.2.2.2 Тестовые примеры на робастность

Тестовые примеры на робастность формируются, чтобы показать, как программное обеспечение ведет себя при воздействии аномальных или неожиданных условий и входов. DO-178C требует тестовых примеров на робастность по требованиям высокого и низкого уровней для уровней А, В и С, а для уровня D – по требованиям высокого уровня. Тестовые примеры на робастность рассматривают недопустимые значения переменных, неверные переходы состояний, вычисленные счетчики циклов вне диапазона, механизмы защиты от нарушений обособления и арифметического переполнения, ненормальную инициализацию системы и отказы входных данных [1]. DO-178C делает сильный акцент на использовании требований как средства определения тестовых примеров. Требования задают предполагаемое поведение, и именно такое поведение нужно исследовать. Хотя поведение и задано, тестирующие должны стремиться исследовать при написании тестовых примеров и иное или дополнительное поведение, опираясь на понимание того, что система должна делать, и пытаясь проверить отсутствие иных вариантов поведения в реализации, то есть применять

подход *сломай это*. Это открытый процесс, поскольку невозможно определить все дополнительные варианты поведения, которые могут существовать. Исходный набор требований служит ориентиром, а любые новые тестовые примеры, добавляемые для проверки поведения сверх заданного, могут приводить к появлению дополнительных требований, например требований к робастности, поскольку все тесты, предъявляемые для сертификационного зачета, должны быть трассируемы к требованиям.

Тестирование служит двум целям: (1) удостовериться, что требования выполняются, и (2) показать отсутствие ошибок. Вторая цель является наиболее сложной, и именно здесь опыт особенно полезен. Следует помнить, что первоначально предложенные требования представляют собой минимальный набор. При разработке тестовых примеров могут быть выявлены отсутствующие или недостаточные требования, особенно требования к робастности. Изменения требований следует предлагать и отслеживать через формальный процесс внесения изменений, чтобы обеспечить их надлежащее рассмотрение.

Во многих проектах при написании тестов надевают *шоры требований*, то есть пишут тесты только по уже записанным требованиям. Это может ускорить тестирование и уменьшить число отказов при прогоне тестов, но для проекта это не обязательно хорошо. Такой подход позволяет ошибкам ускользать, а лишней функциональности – оставаться необнаруженной. Хорошие тестировщики выходят за пределы буквального текста требований настолько, насколько это необходимо, чтобы искать отсутствующие требования, ошибки и непредусмотренную функциональность [11].

9.6.3 Обзор стратегий тестирования

Прочитав десятки книг по тестированию программного обеспечения, я поражаюсь тому, насколько велика пропасть между книжным изложением и реальными проектами по критическому по безопасности ПО. Например, в литературе по программной инженерии широко используются термины *тестирование методом черного ящика* и *тестирование методом белого ящика*. Тестирование методом черного ящика – это функциональное и поведенческое тестирование, выполняемое без знания кода. Тестирование методом белого ящика, иногда называемое *тестированием через стекло*, использует знание внутренней структуры программы и кода для тестирования ПО. Эти термины, особенно *тестирование методом белого ящика*, часто затушевывают необходимую связь с требованиями. Поэтому в DO-178C эти термины не используются. На деле DO-178C в основном избегает стратегий тестирования методом белого ящика. Вместо этого DO-178C сосредоточивается на тестировании требований высокого и низкого уровней, чтобы убедиться, что код правильно реализует требования.

Несмотря на эту пропасть между литературой и DO-178C, многие концепции тестирова-

ния методом черного ящика, а в некоторой степени и тестирования методом белого ящика, можно применять к критическому по безопасности программному обеспечению. По моему опыту, подходы тестирования методом черного ящика применимы и к тестированию по требованиям высокого и низкого уровней. Стратегии тестирования методом белого ящика применять несколько сложнее. Однако если применять их к низкоуровневым требованиям и архитектуре, а не к самому коду, они все же дают полезные практики. Разумеется, для этого требования низкого уровня должны быть написаны на соответствующем уровне. Глоссарий DO-178C определяет требования низкого уровня так: «Требования к ПО, разработанные на основе требований высокого уровня, производных требований и проектных ограничений, по которым исходный код может быть непосредственно реализован без дополнительной информации» [1]. Если рассматривать требования низкого уровня как проектные детали непосредственно над уровнем кода, многие концепции тестирования методом белого ящика можно применять. Но нужно следить за тем, чтобы тесты писались по требованиям, а не по коду.

В этом разделе кратко рассматриваются несколько стратегий тестирования, которые могут применяться в проекте. Обзор намеренно остается общим: его задача – начать сокращать разрыв между концепциями тестирования в DO-178C и тем, что обычно излагается в учебниках по тестированию программного обеспечения.

9.6.3.1 Разбиение на классы эквивалентности

Прессман пишет:

Разбиение на классы эквивалентности относится к тестированию методом черного ящика: оно делит входную область программы на классы данных, из которых можно вывести тестовые примеры. Идеальный тестовый пример в одиночку выявляет класс ошибок, например некорректную обработку всех символьных данных, для обнаружения которого в противном случае могло бы потребоваться выполнение множества тестовых примеров, прежде чем станет заметна общая ошибка [12].

Разбиение на классы эквивалентности требует оценки самих классов эквивалентности. Глоссарий DO-178C определяет *класс эквивалентности* как: «Такое подмножество значений входных данных программы, что тестирование, выполненное для репрезентативного значения из этого класса, эквивалентно тестированию с другими значениями из этого класса» [1]. Тестирование по классам эквивалентности рассматривает классы входных условий, которые необходимо протестировать; при этом каждый класс покрывает большой набор других возможных тестов. Такой подход «позволяет тестирующему систематически оценивать входные или выходные переменные для каждого параметра в некоторой функциональности» [9]. Обычно этот подход применяют в ситуациях, когда для диапазона

выделяются допустимые, то есть нормальные, и недопустимые входы, используемые для проверки робастности. Вместо того чтобы тестировать каждое значение диапазона, достаточно использовать только репрезентативные значения, обычно с обеих сторон границ. Для больших диапазонов данных типичными также являются значение в середине и значения на крайних точках каждого конца диапазона. Помимо диапазонов значений, классы эквивалентности учитывают похожие группы переменных, уникальные значения, требующие отдельной обработки, и специальные значения, которые должны присутствовать или, наоборот, не должны встречаться.

Хотя разбиение на классы эквивалентности может казаться тривиальным, применять его бывает весьма сложно. Его эффективность опирается на «способность тестировщика точно разложить изменяющиеся данные заданного параметра на хорошо определенные подмножества, в которых любой элемент конкретного подмножества логически порождает бы тот же ожидаемый результат, что и любой другой элемент того же подмножества» [9]. Если тестировщик не понимает систему и предметную область, он может пропустить критические неисправности и/или выполнить избыточное тестирование [9].

Следующие рекомендации могут помочь при поиске классов эквивалентности [7,10]:

- учитывайте недопустимые входы. Именно там часто находятся уязвимости программного обеспечения и ошибки. Эти же случаи служат и тестовыми примерами на робастность. Полезно организовать классификацию в виде таблицы с колонками: входное или выходное событие, допустимые классы эквивалентности и недопустимые классы эквивалентности;
- выявляйте числовые диапазоны. Обычно для числового диапазона тестируют значения внутри диапазона, на обеих границах и в середине, значение ниже минимума диапазона, значение выше максимума диапазона и нечисловое значение;
- выявляйте принадлежность к группе;
- учитывайте переменные, которые должны быть равны;
- учитывайте эквивалентные выходные события. Иногда это бывает сложно, потому что нужно определить, какие входы порождают данные выходы.

9.6.3.2 Анализ граничных значений

Анализ граничных значений (boundary value analysis, BVA) – это метод проектирования тестовых примеров, дополняющий разбиение на классы эквивалентности. Вместо выбора произвольного элемента класса эквивалентности этот метод ведет к выбору тестовых примеров на «краях» класса. Вместо того чтобы сосредотачиваться только на входных условиях, он выводит тестовые примеры и из области выходов [12].

Анализ граничных значений тесно связан с разбиением на классы эквивалентности, но имеет два отличия: (1) он проверяет каждую границу класса эквивалентности и (2) он исследует выходы классов эквивалентности [8]. Граничные условия могут быть тонкими и трудными для выявления [8].

Граничные значения – это самые большие, самые маленькие, самые ранние, самые короткие, самые громкие, самые быстрые, самые уродливые представители класса эквивалентности, то есть наиболее экстремальные значения. Также рассматриваются неверные отношения равенства, например $>$ вместо $\geq$ [10]. Программисты могут случайно задать неправильные границы для линейных переменных, поэтому границы являются одним из лучших мест для поиска ошибок. Тестирование границ особенно эффективно для обнаружения ошибок в циклах, ошибок типа *на единицу больше или меньше* и неверных операторов сравнения [9]. Важно учитывать выходы так же, как и входы. Следует также рассматривать и средние значения диапазона.

Классическая ошибка программирования – промахнуться на единицу при задании количества итераций цикла: например, программист может проверить $<$, когда следовало проверять $\leq$. Поэтому тестирование границ циклов может быть хорошим местом для поиска ошибок. При тестировании цикла сначала проверяют обход цикла без входа в него, затем один проход, два прохода, максимально допустимое число проходов, на один проход меньше максимума и на один проход больше максимума. Такой вид тестирования часто выполняется как часть тестирования по требованиям низкого уровня, поскольку циклы обычно являются проектной деталью, а не деталью функциональных требований.

9.6.3.3 Тестирование переходов состояний

Если в системе используются переходы состояний, тестирование переходов состояний рассматривает допустимые и недопустимые переходы между состояниями. Недопустимые переходы состояний, которые могут повлиять на безопасность или функциональность, следует тестировать наряду с допустимыми переходами.

9.6.3.4 Тестирование по таблицам решений

Иногда требования включают таблицы решений для представления сложных логических связей. Поскольку такие таблицы фактически выполняют роль требований, их необходимо тестировать. Обычно условия в таблице интерпретируются как входы, а действия – как выходы. В таблице также могут присутствовать классы эквивалентности. Тестировщики должны показать, что требования, представленные в таблице, полноценно проверены. Тестирование может выявить недостатки самой таблицы, например неверную логику или неполные данные, а также неправильную реализацию в коде.

9.6.3.5 Интеграционное тестирование

Интеграция – это процесс объединения программных компонентов. Компоненты могут прекрасно работать по отдельности при тестировании по требованиям низкого уровня, но их интеграция может дать неожиданные результаты. При интеграции компонентов многое может пойти не так: потеря данных на интерфейсе, неблагоприятное влияние одного компонента на другой, некорректный вклад подфункций в требуемую функциональность более высокого уровня, повреждение глобальных данных и так далее [12]. Чтобы избегать подобных интеграционных препятствий, обычно используют стратегию интеграции. Ниже кратко рассматриваются некоторые распространенные подходы к интеграции.

Интеграция по принципу большого взрыва (big bang). При этом подходе программные компоненты просто собирают вместе, чтобы посмотреть, заработает ли это; обычно не заработает. Хотя здравый смысл подсказывает, что такой подход плох, он все равно встречается. Я видела, как несколько компаний безуспешно пытались так работать. Одна из главных проблем интеграции по принципу большого взрыва состоит в том, что при наличии проблемы ее трудно локализовать, поскольку целостность модулей и характер их взаимодействия заранее неизвестны. Ниже обсуждаются более эффективные стратегии интеграции.

Интеграция сверху вниз. При этом подходе компоненты интегрируют, двигаясь вниз по иерархии, обычно начиная с главного управляющего модуля. Для компонентов, которые еще не интегрированы, используются заглушки. Затем заглушки по одной заменяют реальными компонентами. По мере интеграции каждого компонента проводится тестирование.

Интеграция снизу вверх. При этом подходе компоненты нижнего уровня тестируют с использованием драйверов. Драйверы эмулируют компоненты следующего уровня вверх по дереву. Затем компоненты интегрируют вверх по иерархии.

Комбинация интеграции сверху вниз и снизу вверх, иногда называемая сэндвич-интеграцией или гибридной интеграцией. Нередко интеграцию выполняют по одной ветви за раз. При таком подходе требуется меньше заглушек и драйверов.

Предпочтительный подход зависит от архитектуры и других программных особенностей. Обычно лучше сначала тестировать наиболее критичные и высокорисковые модули, а также ключевую функциональность. Затем функциональность добавляют организованно, а не наваливают сразу всей кучей.

9.6.3.6 Тестирование производительности

Тестирование производительности оценивает рабочие характеристики программного обеспечения во встроенной системе и выявляет узкие места. Тестирование производительности используется, чтобы показать выполнение требований к производительности, например по времени и пропускной способности. Также исследуются гонки и другие проблемы, связан-

ные со временем. Тестирование производительности обычно выполняется на протяжении всего процесса тестирования и может приводить к перепроектированию или изменению настроек оптимизации, если результаты неудовлетворительны. Как отмечалось в разделе 9.5, такое тестирование также часто помогает определить запасы ПО в целевой среде.

9.6.3.7 Другие стратегии

Есть и другие стратегии тестирования, которые могут применяться в зависимости от архитектуры системы и организации требований. Например, тестирование циклов может применяться к простым циклам, вложенным циклам и сцепленным циклам. Если в требованиях используется модель, могут применяться методы тестирования на основе модели; разработка и верификация на основе модели рассматриваются в главе 14. Кроме того, для алгоритмов могут потребоваться специальные тесты, зависящие от природы конкретного алгоритма. Например, при тестировании цифровой реализации фильтра запаздывания первого порядка нужно выполнить достаточное число итераций, чтобы показать, что характеристика первого порядка действительно изгибается, а не выглядит просто как линейный наклон.

9.6.3.8 Измерения сложности

Опыт показывает, что сложный код и взаимодействия сложных участков кода обычно более подвержены ошибкам [9]. Сложный код чаще содержит неисправности, и сопровождать его труднее. Измерения сложности часто применяют к коду на этапе разработки, чтобы предупредить программистов о рискованных конструкциях. Когда это возможно, программное обеспечение перепроектируют или рефакторят, чтобы уменьшить сложность. Стандарты кодирования часто содержат указание измерять и минимизировать сложность, например часто применяется мера цикломатической сложности Маккейба. Однако если проблема сложности кода не устранена, она может быть хорошей подсказкой для тестировщиков: именно там могут скрываться уязвимости. Сложное ПО нужно тестировать жестко, то есть большим числом нормальных тестов и тестов на робастность, потому что сложный код обычно содержит больше ошибок. Ошибки часто просачиваются в него при модификации, поскольку такой код трудно понять и сопровождать.

В одном проекте, через который мне довелось пройти, была особенно сложная функция. Ее называли *хрупким кодом*. После каждой ее модификации приходилось полностью повторно тестировать всю подсистему, потому что влияние изменения было непредсказуемым. В итоге для критической по безопасности системы это оказалось неприемлемо, и код пришлось перепроектировать уже на позднем этапе проекта. Слишком сложный или хрупкий код следует перепроектировать как можно раньше. Исправить его в начале проекта куда менее мучительно, чем отслеживать фантомные симптомы в последний момент.

9.6.3.9 Итоги и характеристики хорошего теста

В разделе 9.6.3 рассмотрен ряд подходов к тестированию. Большинство проектов используют комбинацию нескольких или всех упомянутых выше подходов. Существуют и другие подходы, которые мы могли бы рассмотреть. Канер и соавторы объясняют, что независимо от подхода к тестированию существуют некоторые общие характеристики хорошего теста. Хорошие тестовые примеры удовлетворяют следующим критериям [10]:

- «Он с разумной вероятностью обнаружит ошибку». Тесты проектируются для поиска ошибок, а не просто для доказательства работоспособности. При написании теста нужно думать о том, как программа может отказать.
- «Он не избыточен». Избыточные тесты дают мало пользы. Если два теста ищут одну и ту же ошибку, зачем выполнять оба?
- «Он лучший в своем классе». Наиболее эффективны тесты, которые с наибольшей вероятностью находят ошибки.
- «Он не слишком прост и не слишком сложен». Чрезмерно сложные тесты трудно сопровождать, и по ним трудно локализовать ошибку. Чрезмерно простые тесты часто малоэффективны и дают мало пользы.

9.6.4 Планирование тестирования

В главе 5 обсуждался план верификации ПО (Software Verification Plan, SVP). Он задает, как на проекте в целом будет выполняться верификация, а также поясняет стратегию тестирования и состав документации. Однако помимо плана верификации ПО большинство компаний разрабатывают еще и подробный план тестирования, который часто включают в документ «Тестовые примеры и процедуры верификации ПО» (Software Verification Cases and Procedures, SVCP). Этот документ обычно посвящен не только тестированию: он также включает процедуры рассмотрений и анализов. Но в этом разделе внимание сосредоточено именно на тестовой части документа.

Часть документа «Тестовые примеры и процедуры верификации ПО», относящаяся к планированию тестирования, иногда называется *планом тестирования*. Она описывает более подробно, чем план верификации ПО, среду тестирования: необходимое оборудование, инструменты, связанные с тестированием, и сборку стенда тестирования; организацию и структуру тестовых примеров и процедур; категории тестирования; соглашения по именованию тестов; группирование тестов; стратегию трассируемости тестирования; процесс рассмотрения тестовых данных и контрольные перечни; распределение обязанностей; общие процедуры настройки тестирования; процедуры сборки тестирования; критерии рассмотрения готовности к тестированию; а также план аудита или проверки соответствия

конфигурации стенда тестирования. Обычно это живой документ, который обновляется по мере развития работ по тестированию.[*] По мере развития этих работ в него обычно добавляется перечень всех требований с пометкой о том, какая методика тестирования будет применяться. По мере разработки тестов этот документ обновляется так, чтобы включать сводку всех тестовых примеров и процедур, которые будут выполняться в ходе тестирования, а также матрицы трассируемости, показывающие связь между тестовыми примерами и требованиями и между тестовыми примерами и тестовыми процедурами. Перед выполнением тестирования туда также добавляются оценки требуемого времени и план выполнения тестирования, включая порядок запуска. Документ «Тестовые примеры и процедуры верификации ПО» может также включать подход к регрессионному тестированию.

Планирование тестирования в документе «Тестовые примеры и процедуры верификации ПО» полезно потому, что помогает обеспечить следующее:

- все участники понимают ожидания, включая график, обязанности и назначенные задачи;
- работы по разработке и выполнению тестирования организованы;
- все необходимые тесты разработаны и выполнены;
- отсутствует ненужная избыточность;
- графики формируются реалистично, а не вокруг вымышленной даты, спущенной сверху руководством проекта;
- определены приоритеты;
- все задачи обеспечены исполнителями, а возможный дефицит персонала выявлен заранее;
- определены необходимое оборудование и процедуры;
- ожидаемая организация тестирования доведена до участников;
- тестовые примеры и процедуры разработаны, рассмотрены и прогнаны всухую;
- критерии рассмотрения готовности к тестированию определены заранее, чтобы всем было понятно, чего ожидать;
- порядок выполнения тестирования спланирован так, чтобы тестовые работы выполнялись эффективно;
- определены версии тестовых примеров и тестовых процедур, которые будут использоваться для зачета;

- процедуры настройки тестирования ясны;
- все требования покрыты тестами;
- группирование тестов логично;
- необходимое тестовое оборудование приобретено и корректно настроено;
- все инструменты тестирования находятся под надлежащим управлением;
- тестовые файлы собираются так, чтобы представлять реальное программное обеспечение, которое будет поставляться в эксплуатацию;
- стенды тестирования корректно настроены и представляют целевую среду;
- определен подход к аудиту конфигурации стендов тестирования;
- риски выявлены и снижены.

Документ «Тестовые примеры и процедуры верификации ПО» следует писать так, чтобы он служил проекту, а не просто закрывал пункт в перечне документов. Его следует организовать так, чтобы его было легко обновлять на протяжении проекта. Он дает свидетельство того, что комплекс работ по тестированию организован и полон. Без него трудно подтвердить, что все было выполнено должным образом.

Также обратите внимание: в главе 12, разделе 12.5, поясняется ожидаемая зрелость программного обеспечения перед летными испытаниями, проводимыми для сертификационного зачета. Этот критерий зрелости следует учитывать при планировании тестирования, поскольку он может влиять на график и подход.

9.6.5 Разработка тестов

Выше уже рассмотрены ожидания DO-178C к тестированию и типичные применяемые стратегии. В этом разделе подробнее разбирается разработка тестовых примеров и тестовых процедур с точки зрения DO-178C.

9.6.5.1 Тестовые примеры

DO-178C определяет тестовый пример как «набор тестовых входных данных, условий выполнения и ожидаемых результатов, разработанный с конкретной целью: выполнить тестирование конкретной ветви программы или проверить соответствие конкретному требованию» [1]. Обычно команде тестирования предоставляется шаблон, чтобы тестовые примеры оформлялись единообразно. Если тестирование автоматизировано, единый формат особенно важен. Кроме того, иногда используется инструмент, который извлекает информацию из заголовков тестовых примеров для формирования сводного отчета по тестированию или отчета о трассируемости. Независимо от того, является ли тестирование ручным или автоматизированным, каждый тестовый пример обычно включает следующее:

- идентификатор тестового примера;
- историю его редакций;
- автора теста;
- идентификацию тестируемого программного обеспечения;
- описание теста;
- проверяемое требование или требования;
- категорию теста: по требованиям высокого уровня, по требованиям низкого уровня или одновременно по требованиям высокого и низкого уровней;
- тип теста: нормальный или на робастность;
- тестовые входы;
- шаги или сценарии выполнения теста;
- тестовые выходы;
- ожидаемые результаты;
- критерии прохождения или непрохождения.

9.6.5.2 Тестовые процедуры

Помимо тестовых примеров DO-178C упоминает необходимость тестовых процедур. Тестовая процедура – это «подробные инструкции по подготовке и выполнению заданного набора тестовых примеров, а также инструкции по оценке результатов их выполнения» [1]. Тестовые процедуры бывают разными по форме и объему. Иногда они фактически встроены в сами тестовые примеры; в других случаях это отдельные документы верхнего уровня, объясняющие, как выполнять тестовые примеры. Тестовые процедуры описывают шаги, по которым выполняются тестовые примеры и получаются результаты тестирования. Они могут быть ручными или автоматизированными.

Каждый шаг тестирования и способ определить успех или неуспех должны быть ясными и воспроизводимыми. Человек, выполняющий тест, часто не является его автором. Более того, некоторые компании и сертифицирующие органы настаивают на независимом выполнении тестирования. Неясные тестовые процедуры являются распространенной проблемой; шаги могут быть понятны автору теста, но человек, не знакомый с тестом или с данной функциональностью, может не суметь успешно выполнить тестирование. Чтобы проверить ясность и воспроизводимость, полезно как можно раньше выполнить пробный прогон силами человека, который эти тесты не писал. На деле это может являться частью процесса рассмотрения тестирования, который обсуждается далее.

9.6.5.3 Требования DO-178C

В таблице 9.1 сведены минимальные требования DO-178C к тестированию по требованиям высокого и низкого уровней, а также к нормальному тестированию и тестированию на робастность.

Таблица 9.1 Сводка требований DO-178C к тестированию

	Тесты по требованиям высокого уровня:	Тесты по требованиям высокого уровня: на робастность	Тесты по требованиям низкого уровня:	Тесты по требованиям низкого уровня: на робастность
Уровень ПО	нормальные		нормальные	
Уровень А	Требуются	Требуются	Требуются с независимостью	Требуются с независимостью
Уровень В	Требуются	Требуются	Требуются с независимостью	Требуются с независимостью
Уровень С	Требуются	Требуются	Требуются	Требуются
Уровень D	Требуются	Требуются	Не требуются	Не требуются

Источник: RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Washington, DC, December 2011.

9.6.5.4 Тестирование по требованиям низкого уровня и модульное тестирование

Прежде чем переходить к выполнению тестирования, я хочу кратко обсудить *модульное тестирование*. Некоторые компании используют модульное тестирование (unit testing) для тестирования по требованиям низкого уровня, но с этим нужно обращаться осторожно. Традиционное модульное тестирование выполняется через написание тестов по коду. Это, безусловно, не запрещено и является хорошей практикой для самих разработчиков кода, чтобы убедиться, что код делает то, что они ожидают, то есть для отладки. Однако DO-178C подчеркивает, что тесты, предъявляемые для сертификационного зачета, должны выполняться по требованиям. Подходы модульного тестирования по-прежнему можно использовать, когда каждый модуль или группа модулей тестируется по требованиям низкого уровня. Но нужно следить, чтобы тесты писались по требованиям, а не по коду.[*]

Кроме того, помодульное тестирование может создавать трудности при интеграции. Если тестирование по требованиям низкого уровня выполняется на уровне модулей или функций, нужен конкретный план того, как эти модули будут интегрироваться и тестироваться совместно. Как обсуждалось ранее, интеграция по принципу *большого взрыва* (big bang) не рекомендуется и обычно неприемлема. Также на самом верхнем уровне интеграции может быть трудно подтвердить корректность интерфейсов между программными модулями или функциями. Когда используется тестирование на уровне модулей, обычно требуется дополнительное тестирование интеграции ПО, помимо тестирования интеграции аппаратуры и ПО по требованиям высокого уровня. Когда далее, в разделе 9.7, будут обсуждаться анализы межкомпонентной связности по данным и по управлению, станет еще очевиднее, почему

такое тестирование интеграции ПО необходимо.

9.6.5.5 Обработка требований, которые нельзя протестировать

На этапе разработки тестов может выясниться, что некоторые требования не поддаются тестированию. В одних случаях требование придется переписать, например если оно отрицательное, не содержит допусков или сформулировано неоднозначно. В других случаях вместо тестирования могут использоваться анализ или инспекция кода. К таким ситуациям нужно относиться осторожно, поскольку в общем случае анализ и инспекция кода не считаются такими же эффективными, как тестирование.[*] Если вместо тестирования используется анализ или инспекция кода, необходимо обосновать, почему анализ или инспекция кода выявят те же виды ошибок, что и тестирование, и почему они эквивалентны тестированию или лучше него. Я рекомендую включать такое обоснование прямо в материалы анализа или инспекции кода, чтобы оно было явно зафиксировано. Анализ также должен быть хорошо документирован и воспроизводим, как отмечалось в разделе 9.5. Анализ или инспекция кода, используемые вместо тестирования, требуют того же уровня независимости, который потребовался бы для тестирования.

9.6.5.6 Получение зачета за несколько уровней тестирования

В некоторых ситуациях проект может подготовить часть тестов так, чтобы они одновременно охватывали требования высокого и низкого уровней. Обычно вероятнее, что тест по требованиям высокого уровня сможет подтвердить и требования низкого уровня, а не наоборот. Но это зависит от подхода к интеграции и от степени детализации требований. Если один тест или группа тестов используются для заявления зачета сразу по нескольким уровням требований, необходимо иметь документированные свидетельства того, что удовлетворены оба уровня требований, включая данные трассировки и записи рассмотрений. В некоторых случаях может потребоваться анализ, показывающий, как тест или тесты полностью охватывают оба уровня требований, особенно если тесты изначально не писались для покрытия нескольких уровней требований. Такой анализ следует включать в состав данных тестирования.

9.6.5.7 Тестирование дополнительных уровней требований

В некоторых проектах, помимо требований высокого и низкого уровней, могут существовать и другие промежуточные уровни требований. Эти требования также потребуется тестировать, за возможным исключением программного обеспечения уровня D. Как отмечалось в предыдущем разделе, возможно, удастся использовать один набор тестов, основанных на требованиях, для охвата нескольких уровней требований.

9.6.6 Выполнение тестирования

Тестовые примеры и тестовые процедуры пишутся для того, чтобы их выполнять. В этом разделе рассматриваются некоторые вопросы, которые следует учитывать при подготовке и выполнении тестирования.

9.6.6.1 Проведение пробных прогонов

В большинстве проектов выполняют пробный прогон тестов, чтобы выявить недочеты и устранить проблемы до *зачетных* или *формальных* прогонов, используемых для сертификационного зачета. Пробный прогон тестов настоятельно рекомендуется. Попытка сразу перейти от бумажного рассмотрения к формальному выполнению обычно приводит к неожиданным и нежелательным сюрпризам. Для более высоких уровней программного обеспечения, то есть уровней А и В, хорошей практикой является поручать пробный прогон независимому исполнителю, который не писал процедуры. Это помогает выявить проблемы в процедурах, а также обнаружить неожиданные отказы.

9.6.6.2 Обзор тестовых примеров и процедур

До начала выполнения тестирования тестовые примеры и тестовые процедуры должны быть проанализированы посредством рассмотрения, а для уровней А, В и С также взяты под управление изменениями. Нередко тесты неформально прогоняются еще до рассмотрения, чтобы во время рассмотрения можно было сослаться на результаты. Процесс рассмотрения подробнее рассматривается в разделе 9.7.

9.6.6.3 Использование целевого вычислителя по сравнению с эмулятором или симулятором

Тестирование обычно выполняется на целевом вычислителе, особенно тестирование интеграции аппаратуры и ПО. Иногда для выполнения тестирования используется эмулятор целевого вычислителя или симулятор на базе инструментального компьютера. Если это так, необходимо оценить различия между эмулятором или симулятором и целевым вычислителем, чтобы убедиться, что способность обнаруживать ошибки остается такой же, как на целевом вычислителе. Демонстрация эквивалентности может быть достигнута с помощью анализа различий[*] или посредством квалификации эмулятора либо симулятора, используя подход к квалификации инструментов, описанный в DO-178C и DO-330, который подробнее будет рассмотрен в главе 13. Следует отметить, что некоторые тесты, вероятно, все равно придется повторно выполнить на целевом вычислителе, даже если используется эмулятор или симулятор, поскольку существуют виды ошибок, которые могут быть обнаружены только в среде целевого вычислителя.[†],[‡]

9.6.6.4 Документирование среды верификации

Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index, SECI) или эквивалентный документ должен быть актуализирован до выполнения тестирования для сертификационного зачета.

Этот указатель документирует детали среды верификации и обычно используется для настройки стендов тестирования до запуска тестов на сертификационный зачет. Глава 10 содержит сведения о составе этого указателя.

9.6.6.5 Рассмотрение готовности к тестированию

В DO-178C рассмотрение готовности к тестированию (test readiness review, TRR) не рассматривается; однако большинство проектов проводят такое рассмотрение перед выполнением тестирования, чтобы убедиться в следующем:

- требования, проект ПО, исходный код и исполняемый объектный код помещены в базовую конфигурацию;
- тестирование согласовано с базовой версией требований, проекта ПО и кода;
- тестовые примеры и тестовые процедуры рассмотрены и помещены в базовую конфигурацию; все проблемы, выявленные в ходе пробных прогонов, устранены;
- тестовые сборки завершены, а обозначение проверяемого программного обеспечения задокументировано;
- используется утвержденная тестовая среда;
- стенды тестирования настроены в соответствии с утвержденным указателем конфигурации среды ЖЦ ПО или эквивалентным документом;
- данные трассировки тестирования актуальны; трассируемость будет обсуждаться далее в этой главе;
- график тестирования доведен до тех, кто должен наблюдать тестирование или обеспечивать его выполнение, например до персонала по обеспечению качества программного обеспечения, заказчика и/или специалистов по взаимодействию с сертифицирующим органом;
- все известные сообщения о проблемах (problem report, PR) по проверяемому программному обеспечению либо закрыты, либо согласованы к отсрочке; если ожидается, что некоторые тесты завершатся неуспехом, важно заранее знать об этих ожидаемых неудачах до начала формального прогона.

Следует отметить, что в некоторых крупных проектах тестирование выполняется группами. В такой ситуации для каждой группы тестов может проводиться свое рассмотрение готов-

ности к тестированию. В этом случае нужно следить за тем, чтобы все требования были полностью оценены и чтобы учитывались любые зависимости между группами тестов.

9.6.6.6 Выполнение тестирования для сертификационного зачета

Выполнение тестирования – это исполнение тестовых примеров и тестовых процедур, наблюдение за реакциями и регистрация результатов. Выполнение тестирования часто поручается тому, кто не писал эти тесты, и нередко проходит в присутствии представителей гарантии качества ПО и/или заказчика.

Во время формального выполнения тестирования тестовые процедуры последовательно исполняются, а выполненные шаги отмечаются по мере прохождения. Сами процедуры к этому моменту уже должны быть рассмотрены и помещены в базовую конфигурацию.[*] Обычно ведется журнал тестирования, в котором фиксируется, кто выполнял тест, когда он выполнялся, кто присутствовал при его выполнении, сведения о версиях тестов и программного обеспечения, а также результаты тестирования. Нередко решение об успехе или неуспехе теста принимается прямо во время его выполнения. Однако если данные требуют анализа, окончательное решение об успехе или неуспехе может быть принято позже.

Когда тестирование выполняется для сертификационного зачета, то есть в ходе формального прогона, в процедуры могут вноситься незначительные *исправления по месту* [*]. Если пробный прогон был проведен качественно, таких исправлений должно быть немного. Слишком большое количество исправлений обычно указывает на незрелость тестовых процедур. Если исправлений немного, их обычно фиксируют на бумажной копии или в журнале тестирования, а затем, до выпуска результатов, их утверждают инженерный персонал и служба гарантии качества ПО. Данные с внесенными исправлениями, как правило, утверждаются службой гарантии качества ПО и включаются в сообщение о проблеме, чтобы позже обновить тесты.

При выполнении для сертификационного зачета тесты могут выявить неожиданные отказы. В таких случаях отказы анализируются и принимаются соответствующие меры. Нередко тесты и/или требования обновляются, после чего тесты выполняются повторно. Отказы тестирования и последующие обновления должны обрабатываться через процесс регистрации сообщений о проблемах. Для повторно выполняемых тестов может потребоваться анализ регрессии. Дополнительное обсуждение процесса регрессионного тестирования приведено в разделе 9.6.9.

9.6.7 Отчетность по тестированию

После выполнения тестирования результаты анализируются и обобщаются. Любой отказ теста должен приводить к оформлению сообщения о проблеме и должен быть проанализирован. Рассмотрение отказов тестов обсуждается в разделе 9.7.

Обычно для обобщения результатов работ по верификации формируется документ «Результаты верификации ПО» (Software Verification Results, SVR). Он охватывает все работы по верификации, включая рассмотрения, анализы и тестирование, а также их результаты. Нередко он также содержит окончательные данные трассируемости.

Для любых работ по верификации, которые завершились неуспешно, документ SVR обычно кратко излагает анализ из сообщения о проблеме и обосновывает, почему эти отказы приемлемы.

Результаты верификации ПО обычно рассматриваются сертифицирующим органом и/или его уполномоченными представителями, а также заказчиком. Поэтому этот документ следует оформлять в расчете на аудиторию, которая может не быть знакома с подробностями разработки и верификации.

9.6.8 Трассируемость тестирования

В разделе 6.5 DO-178C подчеркивается, что должны существовать данные трассировки, демонстрирующие двунаправленные связи между всеми следующими элементами [1]:

- требования к программному обеспечению и тестовые примеры;
- тестовые примеры и тестовые процедуры;
- тестовые процедуры и результаты тестирования.

Эти данные трассировки часто включаются в результаты верификации ПО, но могут быть включены и в документ «Тестовые примеры и процедуры верификации ПО» (Software Verification Cases and Procedures, SVCP) либо в отдельный документ. Точность данных трассировки должна проверяться при рассмотрении тестовых примеров, тестовых процедур и результатов тестирования. В некоторых случаях трассировка неявная, поскольку документы объединены: например, тестовые примеры и тестовые процедуры могут находиться в одном и том же документе. В других случаях может использоваться соглашение об именовании; например, каждая тестовая процедура может порождать результат тестирования с тем же именем, например TEST1.CMD и TEST1.LOG. Если применяется такой подход, стратегию следует объяснить в плане верификации ПО (Software Verification Plan, SVP) и кратко изложить в результатах верификации ПО.

9.6.9 Регрессионное тестирование

Регрессионное тестирование – это повторное выполнение части или всех тестов, разработанных для конкретного мероприятия по тестированию [8]. После выполнения тестирования для сертификационного зачета могут потребоваться дополнительные тесты. Это могут быть тесты, которые завершились неуспешно при первоначальном прогоне и были изменены, либо новые или измененные тесты, добавленные из-за новой или измененной функци-

ональности.

Прессман поясняет, что регрессионный прогон тестов обычно включает следующие виды тестов [12]:

- репрезентативную выборку тестов, которая задействует все функции программного обеспечения, чтобы поддержать регрессионное тестирование побочных эффектов и устойчивости; тесты, сфокусированные на самом изменении;
- тесты, сфокусированные на потенциальных последствиях изменений программного обеспечения, то есть тесты затронутых элементов данных.

До проведения такого тестирования необходим анализ влияния изменений. Как минимум анализ влияния изменений учитывает данные трассировки, а также потоки данных и управления, чтобы выявить все измененные и затронутые элементы данных, например требования, проект ПО, код, тестовые примеры и тестовые процедуры. Анализ влияния изменений также определит, какая именно повторная верификация требуется.

Повторная верификация обычно включает рассмотрения обновленных и затронутых данных, переработку всех затронутых анализов, повторный прогон новых, измененных и затронутых тестов, а также выполнение описанного выше регрессионного тестирования. Глава 10 дает дополнительное представление о том, что обычно учитывается при анализе влияния изменений.

В зависимости от масштаба изменения иногда разумнее повторно выполнить все тесты, чем пытаться обосновать частичный регрессионный подход. Особенно это справедливо для автоматизированного тестирования. Тщательный анализ влияния изменений может занять значительное время. Порой проще повторно выполнить тесты, чем обосновывать, почему их повторный запуск не требуется.

Если принимается регрессионный подход и для отработки изменения выполняется небольшое число тестов, разумно также запускать подмножество общего набора тестов, чтобы подтвердить отсутствие непреднамеренных изменений. Некоторые проекты при каждом изменении запускают стандартное подмножество общего набора тестов в дополнение к тестам измененного или затронутого программного обеспечения. Такое подмножество обычно включает тесты, связанные с безопасностью, и тесты, подтверждающие критические функции. Иногда это называют *регрессией по побочным эффектам* или *регрессией устойчивости*, поскольку такие тесты подтверждают, что изменение не внесло побочного эффекта и не повлияло на устойчивость кодовой базы. Такое регрессионное тестирование помогает убедиться, что изменения не вносят непреднамеренное поведение или дополнительные ошибки.

9.6.10 Пригодность к тестированию

Пригодность к тестированию следует учитывать в ходе разработки. В главе 7 объясняется, что пригодное к тестированию программное обеспечение является работоспособным, наблюдаемым, управляемым, декомпозируемым, простым, стабильным и понятным. В главе 7 также указаны особенности, которые помогают сделать ПО пригодным к тестированию, включая журналирование ошибок или отказов, средства диагностики, тестовые точки и доступные интерфейсы. Участие специалистов по тестированию в рассмотрении требований и проекта ПО может помочь сделать ПО более пригодным к тестированию.

9.6.11 Автоматизация в процессах верификации

Автоматизация широко применяется при выполнении работ по верификации. Инструменты не заменяют необходимость думать, и пользователи должны понимать, как эти инструменты работают, чтобы применять их эффективно; однако есть задачи, которые инструменты выполняют эффективнее и точнее человека. Ниже приведены некоторые примеры автоматизации, используемой в процессах верификации:

- *Шаблоны тестовых документов* обычно используются для плана-графика и задач тестирования, плана тестирования, тестовых примеров, тестовых процедур и отчета по тестированию. Степень автоматизации здесь сильно различается.
- *Инструменты тестовых сценариев* предоставляют язык для написания автоматизированных тестов, а также шаблон, который поддерживает единообразную структуру тестов.
- *Инструменты трассируемости* применяются для сбора и проверки данных трассировки. Сами данные в инструменте все равно нужно проверять, но такие инструменты помогают выявлять отсутствующие тестовые примеры или тестовые процедуры, отсутствующие результаты, непроверенные требования и тому подобное.
- *Инструменты выполнения тестов* исполняют тестовые сценарии и могут использоваться для определения результата тестирования, то есть прохождения или неуспеха. Если выходные данные тестов не анализирует человек, инструмент, возможно, придется квалифицировать, поскольку он автоматизирует выполнение целей DO-178C. Критерии квалификации инструментов рассматриваются в главе 13.
- *Отладочные инструменты* иногда применяются в ходе тестов для установки точек останова, чтобы исследовать программное обеспечение или воздействовать на него. Во время формальных тестов их следует использовать осторожно, чтобы они не изменяли ПО нежелательным образом. Кроме того, избыточное количество точек останова может затруднить доказательство полноты интеграции. В зависимости от того, как

именно используется отладочный инструмент, его функциональность может потребовать квалификации.

- *Инструменты работы с памятью* используются для выявления проблем с памятью: перезаписи памяти, выделенной, но не освобожденной памяти, а также использования неинициализированной памяти [8].
- *Эмуляторы или симуляторы* иногда используются в процессе тестов для доступа к внутренним данным, к которым на реальной целевой платформе может быть трудно получить доступ.
- *Инструменты покрытия* помогают при оценке анализа структурного покрытия. Такие инструменты часто инструментируют код, чтобы собирать метрики по его выполнению. Поэтому может потребоваться запускать тесты как с инструментированием, так и без него, сравнивать результаты и убеждаться, что инструментирование не влияет на результаты тестирования.
- *Инструменты статического анализа* используются для оценки сложности кода, соответствия правилам, наихудшего пути выполнения, наихудшего времени выполнения, наихудшего использования стека и так далее. Инструменты статического анализа также могут оценивать потоки данных и управления, выявляя неопределенные переменные, несогласованные интерфейсы между модулями и неиспользуемые переменные [7].
- *Генераторы тестовых векторов* используют формальные описания или модели. Такие инструменты генерируют тестовые векторы для проверки точности реализации. В зависимости от способа их использования они могут требовать квалификации. Также рекомендуется дополнять сгенерированные инструментом тестовые векторы тестами, созданными человеком, чтобы полноценно проверить функциональность. Важно, чтобы генераторы тестовых векторов брали в качестве входных данных требования, а не код. Существуют инструменты, которые генерируют *требования* из исходного кода, а затем создают тестовые векторы для этих требований. Таких инструментов следует избегать.

9.7 Верификация верификации

Таблица А-7 DO-178С озаглавлена «Результаты процесса верификации». Вероятно, это самая обсуждаемая и спорная таблица в DO-178С, главным образом потому, что она включает несколько целей, характерных именно для авиационной отрасли и редко подробно описанных в другой литературе по программной инженерии, например модифицированное покрытие условий/решений. По существу, цели таблицы А-7 DO-178С требуют от проекта оценить, достаточно ли полон его комплекс работ по тестированию. Иными словами, тре-

будет выполнено *верификация самих мероприятий по тестированию*. Цели таблицы А-7 DO-178С охватывают тестовые процедуры и их результаты, покрытие требований, а также покрытие структуры кода. Каждый из основных аспектов *верификации результатов процесса верификации* рассматривается в этом разделе. На рисунке 9.1 приведен обзор типового процесса.

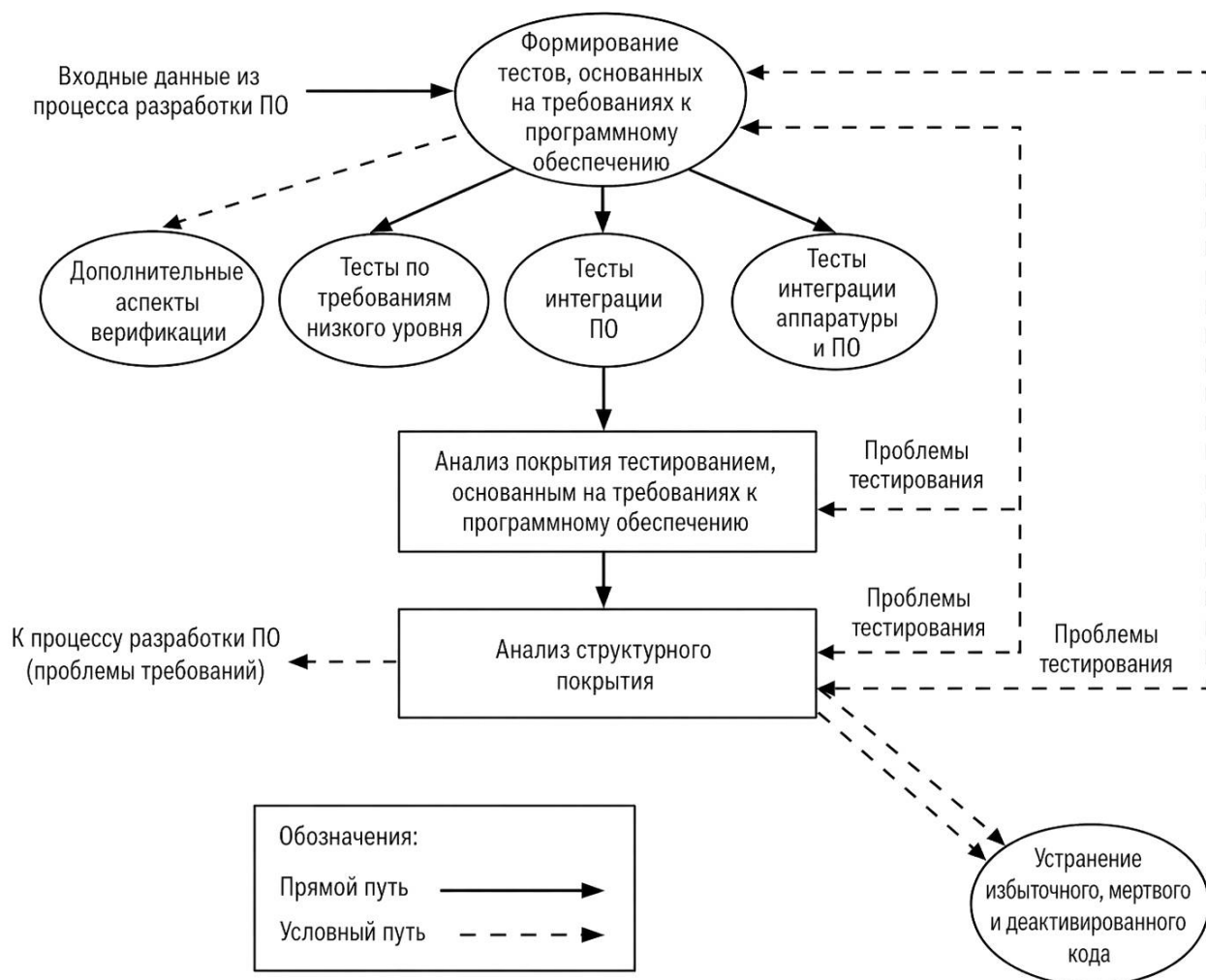


Рисунок 9.1 Обзор тестирования программного обеспечения по DO-178C. (Адаптировано из RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Washington, DC, December 2011, DO-178C Figure 6-1. Использовано с разрешения RTCA, Inc.)

9.7.1 Рассмотрение тестовых процедур

Цель 1 таблицы А-7 DO-178С кратко формулируется так: «Тестовые процедуры корректны» [1]. Цель состоит в том, чтобы убедиться, что тестовые примеры корректно превращены в тестовые процедуры. Обычно это обеспечивается рассмотрением как тестовых примеров, так и тестовых процедур. Некоторые утверждают, что данная цель охватывает только рассмотрение тестовых процедур, поскольку тестовые примеры рассматриваются в рамках целей 3 и 4 таблицы А-7 DO-178С. Однако, по моему опыту, эта цель достигается одновременно с целями 3 и 4, чтобы гарантировать, что все требования проверяются тестированием, что

требования проверяются полноценно, а тестовые примеры и тестовые процедуры являются точными, согласованными и соответствуют плану верификации ПО.

Цель 1 таблицы А-7 DO-178С относится как к тестированию по требованиям высокого уровня, так и к тестированию по требованиям низкого уровня и требуется для уровней А, В и С. Для уровня А требуется независимость. Обычно для выполнения этой цели используется экспертное рассмотрение, аналогичное тому, что применяется при рассмотрении требований, проекта ПО и кода. Сведения о процессе экспертного рассмотрения приведены в главе 6. Типичными входными данными для такого рассмотрения являются требования и проект ПО, план верификации ПО, документ «Тестовые примеры и процедуры верификации ПО», предварительные результаты тестирования (это не то же самое, что пробный прогон), тестовые примеры, тестовые процедуры и данные трассировки. Если вместо какого-либо тестирования используются анализы или инспекции кода, они также предоставляются. Нередко кто-то из команды экспертного рассмотрения запускает тест, чтобы убедиться, что он дает те же результаты, которые представлены на рассмотрении, является повторяемым и дает корректные результаты. Для ручного тестирования это особенно эффективно при устранении неоднозначностей в тестовых процедурах и критериях прохождения/непрохождения. Очевидно, что экспертное рассмотрение должно состояться до выполнения тестирования для сертификационного зачета.

Ниже перечислены наиболее распространенные проблемы, выявляемые в ходе таких рассмотрений:

- некоторые требования не проверяются тестированием;
- тестирование на робастность недостаточно;
- тестирование не обеспечивает полного охвата требований;
- некоторые тесты недостаточно хорошо пояснены или прокомментированы, что затрудняет сопровождение;
- некоторые шаги в процедурах отсутствуют либо сформулированы неясно;
- для некоторых тестов неясны критерии прохождения/непрохождения;
- тесты не запускались для подтверждения их работоспособности;
- трассируемость от требований к тестам не задокументирована, задокументирована только трассируемость от тестов к требованиям;
- среда тестирования задокументирована недостаточно хорошо.

9.7.2 Рассмотрение результатов тестирования

После выполнения тестирования его результаты рассматриваются и анализируются. Цель 2 таблицы А-7 DO-178С кратко формулируется так: «Результаты тестирования корректны, а расхождения объяснены» [1]. Цель состоит в том, чтобы убедиться в корректности результатов тестирования и подтвердить, что все непройденные тесты проанализированы и надлежащим образом обработаны. Обычно эта цель достигается посредством рассмотрения результатов тестирования и результатов верификации ПО. Рассмотрение включает оценку объяснений для любых непройденных тестов, которые обычно приводятся в сообщениях о проблемах и в результатах верификации ПО. В некоторых ситуациях непройденные тесты потребуют исправления программного обеспечения либо тестовых примеров или тестовых процедур.

В общем случае, если был проведен пробный прогон, количество неожиданных отказов тестов сводится к минимуму. Стремясь сократить график, некоторые команды пытаются отказаться от пробных прогонов и сразу перейти к формальному тестированию. Однако проекты, которые пытаются перейти к формальному тестированию без тщательного пробного прогона, обычно в итоге проводят формальное тестирование как минимум дважды. В любом случае все отказы тестов должны быть задокументированы в сообщениях о проблемах и надлежащим образом обработаны. Анализ отказов тестов также обычно включается в результаты верификации ПО.

9.7.3 Анализ покрытия требований

Цели 3 и 4 таблицы А-7 DO-178С называют *целями покрытия требований*. Цель 3 кратко формулируется так: «Достигнуто покрытие тестированием требований высокого уровня» [1]. Аналогично цель 4 формулируется так: «Достигнуто покрытие тестированием требований низкого уровня» [1]. Назначение этих целей состоит в том, чтобы подтвердить, что все требования были проверены тестированием либо верифицированы другим способом при наличии надлежащего обоснования. Обычно эти цели оцениваются в ходе рассмотрения тестовых примеров и тестовых процедур. Однако непосредственно перед или сразу после финального прогона тестов часто выполняется заключительный анализ покрытия, чтобы убедиться, что все требования охвачены и что на завершающих этапах работ по тестированию ничего не упущено, например тестовый пример после позднего изменения требования. Для подтверждения такого покрытия обычно используются данные трассировки между тестовыми примерами и требованиями. Однако также должна выполняться техническая оценка, подтверждающая, что тестовые примеры полностью покрывают требования. Обычно эта оценка проводится в ходе рассмотрения тестовых примеров.

По сути, если тестовые примеры уже были должным образом проанализированы на полное

покрытие требований, сам анализ покрытия становится довольно простым. Если для выполнения тестирования с сертификационным зачетом используется автоматизированный инструмент, важно проверить, что инструмент действительно выполнил все тесты и что ни один из них не был пропущен. Такая проверка может выполняться вручную либо быть автоматизирована в составе инструмента выполнения тестов. Если для этой проверки засчитывается результат работы инструмента, то инструмент должен быть квалифицирован. Поскольку требования низкого уровня не обязаны проверяться тестированием для уровня D, покрытие требований низкого уровня для уровня D не применяется. По моему опыту, анализ покрытия либо включается в результаты верификации ПО, либо кратко излагается в них.

9.7.4 Анализ структурного покрытия

Оставшиеся пять целей таблицы A-7 DO-178C относятся к структурному покрытию. Цели 5-7 и 9 относятся к покрытию структуры ПО тестированием, чтобы убедиться, что код был в достаточной степени выполнен в ходе тестов, основанных на требованиях. Цель 8 аналогична, но сосредоточена на покрытии связности по данным и управлению между компонентами кода, например модулями.

Структурное покрытие преследует следующие цели:

- подтверждает, что весь код был выполнен хотя бы один раз;
- помогает находить непреднамеренную и непроверенную функциональность; выявляет мертвый код[*] или посторонний код[†];
- помогает подтвердить, что отключенный код действительно отключен;
- помогает определить минимальный набор комбинаций для тестов, то есть не требует исчерпывающего тестирования;
- помогает выявлять ошибочную логику;
- служит объективным критерием завершенности работ по тестированию, хотя может и не охватывать полноту тестов на робастность.

Следует отметить, что *анализ структурного покрытия* в DO-178C не эквивалентен *структурному тестированию*, о котором говорится во многих публикациях по программной инженерии. *Анализ* структурного покрытия выполняется для выявления любых элементов структуры кода, которые не были выполнены в ходе тестов, основанных на требованиях. *Структурное тестирование* представляет собой процесс написания тестов на основе кода для выполнения программного обеспечения. Для DO-178C структурного тестирования недостаточно, поскольку оно не основано на требованиях и, следовательно, не удовлетворяет назначению анализа структурного покрытия.

Прежде чем подробнее рассматривать критерии структурного покрытия, уместно сделать краткое предупреждение. Некоторые команды заблаговременно запускают свои инструменты структурного покрытия на всем протяжении разработки тестовых примеров и процедур. Это хорошая практика. Однако некоторые команды используют данные покрытия для определения того, какие тесты нужны, то есть пишут тесты ради удовлетворения инструмента, вместо того чтобы позволить требованиям определять состав тестов. Это не соответствует назначению структурного покрытия и может привести к недостаточному тестированию. Для некоторого ПО покрытие превышает 50 процентов уже просто при включении системы, без запуска хотя бы одного теста. Поэтому необходимо подчеркнуть: тесты должны быть написаны и проанализированы относительно требований до того, как будут собраны данные по структурному покрытию. Чтобы избежать соблазна просто *угодить инструменту*, за анализ покрытия обычно отвечает другой инженер. Заодно это помогает выполнить требование независимости для целей структурного покрытия уровней А и В.

Каждая цель структурного покрытия таблицы А-7 DO-178С кратко рассматривается в следующих подразделах.

9.7.4.1 Покрытие операторов (цель 7 таблицы А-7 DO-178С)

Покрытие операторов требуется для уровней А, В и С; оно подтверждает, что «каждый оператор программы был вызван хотя бы один раз» [1]. Это достигается путем оценки покрытия кода в ходе тестов, основанных на требованиях, и подтверждения того, что каждый исполняемый оператор был вызван хотя бы один раз. Покрытие операторов считается относительно слабым критерием, поскольку оно не оценивает некоторые управляющие конструкции и не выявляет определенные типы логических ошибок. Для конструкции *if-then* достаточно, чтобы решение однажды приняло значение true, чтобы обеспечить покрытие операторов. Для конструкции *if-then-else* требуется, чтобы решение было вычислено как в true, так и в false. По существу, покрытие операторов подтверждает, что код был выполнен, но мог быть выполнен не по той причине.

9.7.4.2 Покрытие решений (цель 6 таблицы А-7 DO-178С)

Покрытие решений требуется для уровней А и В и обычно отождествляется с покрытием ветвей или путей. Однако покрытие решений несколько отличается. DO-178С определяет *решение* и *покрытие решений* следующим образом [1]:

- «Решение. Логическое выражение, составленное из одного или более условий, связанных логическими операциями. Если некоторое условие появляется в решении более чем один раз, каждое его появление считается отдельным условием».
- «Покрытие решений. Каждая точка входа и выхода в программе пройдена хотя бы раз, а также каждое решение в программе приняло все возможные значения хотя бы один

раз».

Путаница возникает вокруг традиционного отраслевого понимания *точки ветвления*, которой обычно является оператор *if-then-else*, *do-while* или *case*, и буквального определения *решения* в DO-178C. По существу, при буквальном толковании DO-178C решение не является синонимом точки ветвления. Главное различие состоит в том, что булево значение, выводимое на порт ввода-вывода (I/O, ввод-вывод), может по-разному влиять на поведение системы, например *шасси убрано* при TRUE и *шасси выпущено* в противном случае. Важно удостовериться, что при тестировании это значение оценивается и как TRUE, и как FALSE. Это можно рассматривать либо как задачу проверки граничных значений, либо как задачу покрытия решений.

В общем случае покрытие решений подтверждает, что в коде проходятся все пути, а если имеются какие-либо булевы присваивания, то должны быть выполнены и истинное, и ложное состояния либо когда значение непосредственно используется вне компонента, либо когда оно используется в конструкции, приводящей к ветвлению.

9.7.4.3 Модифицированное покрытие условий/решений (цель 5 таблицы A-7 DO-178C)

Модифицированное покрытие условий/решений (MC/DC, modified condition/decision coverage) требуется для программного обеспечения уровня А и на протяжении почти всей моей карьеры было предметом многочисленных споров и, вероятно, будет вызывать жаркие дискуссии и после моего выхода на пенсию.

DO-178C определяет *условие*, *решение* и *модифицированное покрытие условий/решений* следующим образом [1]:

- «Условие. Логическое выражение, не содержащее логических операций за исключением унарной операции “Не” (“NOT”)».
- «Решение. Логическое выражение, составленное из одного или более условий, связанных логическими операциями. Если некоторое условие появляется в решении более чем один раз, каждое его появление считается отдельным условием».
- «Модифицированное покрытие условий/решений. Каждая точка входа и выхода в программе пройдена хотя бы раз, каждое условие в каждом решении в программе приняло все возможные значения хотя бы раз, каждое решение в программе приняло все возможные значения хотя бы один раз, а также показано, что каждое условие в решении независимо влияет на исход данного решения. Независимое влияние условия на исход решения демонстрируется посредством: (1) изменения только этого условия при неизменных всех других возможных условиях или (2) изменения только этого условия при неизменных всех других возможных условиях, которые могут влиять на исход».

Хотя критерий MC/DC применяется в авиационных проектах еще с конца 1980-х и начала 1990-х годов, в основной литературе по программной инженерии он по-прежнему обсуждается редко. Он был разработан для авиационной отрасли как способ получить достаточно строгий критерий завершенности тестов, не требуя при этом, чтобы каждая возможная комбинация входов решения была выполнена хотя бы один раз, то есть без исчерпывающего тестирования комбинаций входов решения, известного как *покрытие множественных условий*. Хотя покрытие множественных условий могло бы дать наиболее полный показатель структурного покрытия, во многих случаях оно непрактично, потому что для решения с n входами требуется 2^n тестов [13]. MC/DC, напротив, обычно требует минимум $n + 1$ тестовых примеров для решения с n входами.

Большинство проектов применяют MC/DC на уровне исходного кода, потому что инструменты покрытия исходного кода доступны шире. Однако были и проекты, где критерии структурного покрытия применялись на уровне объектного кода или исполняемого объектного кода. Вокруг такого подхода шли споры, о чем говорится в позиционном документе CAST-17 группы Certification Authorities Software Team (CAST) под названием *Structural coverage of object code*. В этом документе кратко изложены мотивы покрытия объектного кода и некоторые вопросы, которые необходимо учитывать [14]. Документ CAST-17 послужил основой для проектно-специфичных документов issue paper Федерального управления гражданской авиации США (FAA, Federal Aviation Administration). В разделе 6.4.4.2.b DO-178C формулировка была изменена по сравнению с DO-178B так, чтобы явно показать допустимость покрытия объектного кода или исполняемого объектного кода: «Анализ структурного покрытия может выполняться по исходному коду, объектному коду или исполняемому объектному коду» [1]. Часто задаваемый вопрос N42 в DO-248C, озаглавленный «Что необходимо учитывать при выполнении структурного покрытия на уровне объектного кода?», также проясняет эту тему [3].

Очень хочется написать о структурном покрытии, особенно о MC/DC, еще больше, поскольку за последние годы я потратила немало времени на его исследование, обсуждение и оценку. Однако в авиационном сообществе эта тема уже настолько широко обсуждалась, что существует достаточно общедоступных материалов, хорошо ее освещающих. Особенно полезные ресурсы перечислены в пунктах 1-5 раздела «Рекомендуемое чтение» этой главы.

9.7.4.4 Дополнительная верификация кода (цель 9 таблицы A-7 DO-178C)

Цель 9 таблицы A-7 DO-178C кратко формулируется так: «Достигнута верификация дополнительного кода, который не трассируется к исходному коду» [1].

Эта цель была добавлена в DO-178C, хотя соответствующее требование по сути присутствовало и в тексте DO-178B. Большинство проектов выполняют структурное покрытие

на уровне исходного кода, хотя некоторые действительно выполняют покрытие объектного кода или машинного кода. Однако в полете реально работает именно исполняемый объектный код. Для программного обеспечения уровня А требуется анализ, подтверждающий, что компилятор не генерирует нетрассируемый или недетерминированный код.

Для проектов, которые выполняют структурное покрытие на уровне исходного кода, обычно проводится анализ трассируемости от исходного кода к объектному. По моему опыту, он включает сравнение исходного кода и объектного кода, чтобы удостовериться, что компилятор не добавляет, не удаляет и не искажает код. Обычно анализ выполняется по выборке реального кода, а не по 100 процентам кодовой базы.[*] Используемая выборка должна включать все конструкции, разрешенные в исходном коде, и составлять не менее 10 процентов реальной кодовой базы. Также рекомендуется оценивать комбинации таких конструкций. Анализ следует выполнять с теми настройками компилятора, которые будут использоваться при реальной генерации кода. Обычно он включает построчное сравнение исходного кода и объектного кода или машинного кода. Любой объектный код или машинный код, который нельзя напрямую трассировать к исходному коду, анализируется и должен быть объяснен. В некоторых ситуациях анализ может выявить неприемлемое поведение компилятора, и тогда потребуются корректирующие действия.

Такие возможности компилятора, как отслеживание регистров, планирование инструкций и оптимизация ветвлений, могут создавать проблемы для анализа. Сильно оптимизирующие настройки компилятора и некоторые языки, например языки с объектно-ориентированными возможностями, тоже могут приводить к появлению нетрассируемого кода, а значит, такой код может оказаться непригодным для сертификации. Для выполнения анализа нужен инженер, понимающий конкретный язык, ассемблер, машинный код и работу компилятора. Если исходный код меняется, потребуется оценка того, не добавились ли новые конструкции кода и остается ли анализ действительным. После выполнения такого анализа его можно использовать на нескольких проектах, пока не меняются компилятор и его настройки и пока код использует те же конструкции.

В большинстве проектов, в которых мне доводилось участвовать, анализ трассируемости от исходного кода к объектному не выявлял серьезных проблем, если уровень оптимизации оставался минимальным и использовались зрелые языки и компиляторы. Однако этот анализ все равно необходим и должен выполняться тщательно.

9.7.4.5 Анализ межкомпонентной связности по данным и по управлению (цель 8 таблицы А-7 DO-178С)

Самую интересную тему в этом разделе о структурном покрытии я оставила напоследок. Цель по межкомпонентной связности по данным и по управлению (data coupling and control coupling, DC/CC) из таблицы А-7, цель 8, присутствовала уже в DO-178В, но была сфор-

мулирована неясно. В частности, DO-178B расходился от общепринятых в программной инженерии определений межкомпонентной связности по данным и по управлению, но при этом недостаточно ясно объяснял намерение. По словам участников специального комитета RTCA No. 167, разрабатывавшего DO-178B, цель по межкомпонентной связности по данным и по управлению была добавлена на позднем этапе обсуждений в комитете и подробно не обсуждалась. В результате возникла путаница относительно того, что именно подразумевалось под этой целью. К счастью, в DO-178C это уточнили. Однако такое уточнение, вероятно, создаст для отрасли определенные трудности, потому что многие компании не выполняют цель в ее уточненном виде, то есть их трактовка исходной цели DO-178B не совпадает с разъяснением DO-178C.

DO-178C определяет *межкомпонентную связность по данным, межкомпонентную связность по управлению* и *компонент* следующим образом [1]. В самих формулировках стандарта используются краткие нормативные названия «связность по данным» и «связность по управлению»:

- «Связность по данным. Зависимость программного компонента от данных, которые не находятся исключительно под управлением этого программного компонента».
- «Связность по управлению. Способ или степень, с которой один программный компонент влияет на выполнение другого программного компонента».
- «Компонент. Самодостаточная часть, комбинация частей, подузлов или единиц, выполняющая отдельную функцию системы».

Согласно DO-178C, назначение цели 8 таблицы A-7 состоит в выполнении «анализа для подтверждения того, что тесты, основанные на требованиях, выполнили связность по данным и управлению между компонентами кода» [1]. В CAST-19 говорится, что назначение анализа межкомпонентной связности по данным и по управлению заключается в следующем:

Обеспечить измерение и подтверждение корректности взаимодействий и зависимостей между этими модулями/компонентами. Иными словами, намерение состоит в том, чтобы показать, что программные модули/компоненты влияют друг на друга именно теми способами, которые предусмотрел разработчик программного обеспечения, и не влияют друг на друга способами, которые не предполагались и тем самым приводят к незапланированному, аномальному или ошибочному поведению. Обычно такое измерение и подтверждение следует проводить на R-BT [requirements-based tests, тестах, основанных на требованиях] интегрированных компонентов, то есть на итоговой сборке ПО, чтобы гарантировать корректность взаимодействий и зависимостей, полноту покрытия и выполнение цели [16].[*]

К сожалению, именно так критерии понимают далеко не все разработчики. Многие рассматривают их как мероприятие по рассмотрению проекта ПО и кода, призванное удостовериться, что код точно реализует потоки управления и данных, заложенные в проекте ПО. Такое мероприятие в ходе разработки важно, чтобы итоговый анализ межкомпонентной связности по данным и по управлению оказался успешным, а также чтобы выполнить цели таблиц А-4 и А-5 DO-178С; однако в этом не состоит смысл цели таблицы А-7. Помимо рассмотрений проекта ПО и кода, тем, кто соблюдает DO-178С, также нужно удостовериться, что их тесты покрывают межкомпонентную связность по данным и по управлению между компонентами, например модулями или функциями. Некоторые организации применяют критерии межкомпонентной связности по данным и по управлению именно так, как задумывалось, и это может оказаться довольно непросто, особенно если рассмотрения проекта ПО и кода были слабыми или если подход к интеграции выстроен плохо. Если тесты, основанные на требованиях, и измерения структурного покрытия проводятся на интегрированной системе, а не по модулю за модулем, и при этом код не инструментруется, это повышает доверие к результату.

Чтобы успешно выполнить анализ межкомпонентной связности по данным и по управлению с использованием тестов, основанных на требованиях, важно еще в ходе разработки обеспечить согласованность архитектуры с требованиями и соответствие кода архитектуре. Выполнение указаний DO-178С по такому анализу обычно сводится к четырем шагам, описанные ниже.

1. Во-первых, архитектура программного обеспечения должна быть описана (цель 3 таблицы А-2 DO-178С). В разделе 11.10 DO-178С приведены указания о том, что должно быть задокументировано в описании проекта ПО, включая следующие аспекты, относящиеся к потокам данных или управления: структуры данных, архитектура ПО, внутренний и внешний ввод/вывод, потоки данных и потоки управления в проекте ПО, процедуры планирования и механизмы межпроцессорного/межзадачного взаимодействия, методы обособления и описания программных компонентов [1]. Процесс проектирования обсуждался в главе 7.
2. Во-вторых, архитектура программного обеспечения и код должны быть проверены рассмотрением и/или анализом на согласованность. Цель 9 таблицы А-4 DO-178С ссылается на раздел 6.3.3 DO-178С, где поясняется, что одно из назначений рассмотрения и/или анализа проекта ПО состоит в том, чтобы обеспечить корректные отношения между «компонентами архитектуры ПО. Это отношение существует через поток данных и поток управления...» [1]. Цель 2 таблицы А-5 DO-178С ссылается на раздел 6.3.4.b DO-178С, где поясняется, что одно из назначений рассмотрения и/или анализа кода состоит в том, чтобы «убедиться, что исходный код соответствует потоку данных и

потоку управления, определенным в архитектуре ПО» [1]. Обычно как мероприятия по рассмотрению/анализу проекта ПО, так и мероприятия по рассмотрению/анализу кода включают подробный контрольный перечень или опросник, позволяющий рассмотреть типовые проблемы межкомпонентной связности по данным и по управлению. В таблице 9.2 приведены примеры вопросов, которые стоит учитывать при таких рассмотрениях и анализах; конкретика будет зависеть от архитектуры, языка, среды и так далее. Это лишь примеры.

Таблица 9.2 Примеры вопросов по межкомпонентной связности по данным и по управлению, которые следует учитывать при рассмотрении и/или анализе проекта ПО и кода

Примеры вопросов по межкомпонентной связности по данным	Примеры вопросов по межкомпонентной связности по управлению
все внешние входы и выходы определены и корректны;	порядок выполнения определен и корректен;
все внутренние входы и выходы определены и корректны;	частота выполнения определена и корректна;
данные типизированы корректно и согласованно;	условное выполнение корректно;
единицы измерения согласованы и соответствуют словарю данных;	зависимости выполнения определены и корректны;
словарь данных и код согласованы и оба полны;	последовательность, частота и условия выполнения удовлетворяют требованиям;
данные передаются и принимаются в правильном порядке;	прерывания определены и корректны;
данные используются согласованно;	исключения определены и корректны;
повреждение данных предотвращается или обнаруживается;	сбросы определены и корректны;
данные инициализируются или считываются до использования;	реакции на прерывания по питанию определены и корректны;
устаревшие или недействительные данные предотвращаются или обнаруживаются;	планировщики переднего плана выполняются в правильном порядке и с нужной частотой;
рассогласования данных или потери данных предотвращаются или обнаруживаются;	фоновые планировщики выполняются и не зависают в бесконечном цикле;
неожиданные значения с плавающей точкой предотвращаются или обнаруживаются;	код согласован с проектом ПО.
параметры передаются корректно;	
глобальные данные и элементы внутри конструкций глобальных данных корректны;	
ввод/вывод корректно получает доступ к внешним источникам;	
все переменные устанавливаются, то есть инициализируются, до использования;	
используются все переменные;	
переполнение и потеря значимости выявляются и корректно обрабатываются;	
локальные и глобальные данные используются корректно;	
индексы массивов используются корректно;	
код согласован с проектом ПО.	

3. В-третьих, разрабатываются интеграционные тесты, основанные на требованиях (таблица A-5 DO-178C). В разделе 6.4.3.b DO-178C поясняется, что интеграционные тесты ПО, основанные на требованиях, выполняются для того, чтобы убедиться, что «компоненты программного обеспечения правильно взаимодействуют друг с другом и удовлетворяют требованиям к ПО и архитектуре ПО» [1]. Такие интеграционные тесты предназначены для выявления следующих видов ошибок [1]: неправильная инициализация переменных и констант, ошибки передачи параметров, повреждения данных, особенно глобальных, неправильная последовательность событий и операций и недостаточная

сквозная численная точность. Некоторые утверждают, что акцент на *тестах, основанных на требованиях* в пояснении DO-178C к анализу межкомпонентной связности по данным и по управлению ослабляет критерий, потому что архитектура тоже должна быть задействована тестами. Однако при рассмотрении проекта ПО должна проверяться согласованность архитектуры и требований. Если это делается, то тесты, основанные на требованиях, должны также охватывать архитектуру. Кроме того, как отмечено в разделе 6.4.3.b DO-178C, архитектуру следует учитывать при анализе межкомпонентной связности по данным и по управлению.

4. В-четвертых, тесты анализируются для подтверждения того, что межкомпонентная связность по данным и по управлению между компонентами охвачена тестами, основанными на требованиях (цель 8 таблицы A-7 DO-178C). Если шаги 1-3 выполнены недостаточно хорошо, шаг 4 будет трудно, а скорее всего невозможно, выполнить надлежащим образом. Насколько мне известно, коммерчески доступных инструментов для анализа межкомпонентной связности по данным и по управлению пока нет. Большинство компаний либо разрабатывают собственный инструмент или инструменты, либо выполняют анализы вручную. Остается надеяться, что в будущем коммерческие инструменты станут лучше и будут расширены так, чтобы лучше помогать при анализе межкомпонентной связности по данным и по управлению.

Пункт 6 в разделе «Рекомендуемое чтение» этой главы дает дополнительные сведения об анализе межкомпонентной связности по данным и по управлению. Кроме того, поскольку обособление тесно связано с межкомпонентной связностью по данным и по управлению, в главе 21 обсуждаются дополнительные соображения, которые стоит учитывать при разработке подхода к выполнению цели 8 таблицы A-7 DO-178C.

9.7.4.6 Устранение пробелов структурного покрытия

При измерении покрытия кода, полученного за счет тестов, основанных на требованиях, могут выявляться пробелы покрытия. Выполняется анализ, чтобы определить причину этих пробелов. Возможны следующие действия:

- если пробел покрытия вызван отсутствующими тестами, добавляются и выполняются дополнительные тестовые примеры;
- если пробел вызван отсутствующим требованием, то есть недокументированной функцией, требования обновляются, а тесты добавляются и выполняются;
- если пробел выявляет мертвый код или посторонний код, мертвый или посторонний код удаляется и выполняется анализ для оценки последствий и необходимости повторной верификации. Обычно определенная повторная верификация действительно требуется. Сведения о мертвом и постороннем коде приведены в главе 17;

- если пробел вызван отключенным кодом, подход к отключению анализируется, чтобы убедиться, что он достаточен и согласуется с требованиями и проектом ПО. Обсуждение отключенного кода приведено в главе 17.

Если пробел нельзя устранить ни одним из перечисленных выше способов, требуется дополнительный анализ, чтобы убедиться, что код верифицирован надлежащим образом и работает так, как задумано. Обычно такой анализ либо включается в результаты верификации ПО, либо кратко излагается в них.

9.7.4.7 Заключительные замечания об анализе структурного покрытия

Прежде чем закрыть тему структурного покрытия, хочу кратко подытожить четыре важных момента.

Во-первых, если заявляется зачет по структурному покрытию, он должен основываться на тестах, основанных на требованиях, которые прошли успешно, то есть удовлетворяют требованиям, к которым они трассируемы. Если тест не прошел, покрытие может оказаться недействительным.

Во-вторых, структурное покрытие обычно измеряется путем инструментирования исходного кода, чтобы определить, какие части исходного кода были выполнены в ходе тестирования. По мере изменения исходного кода важно оценивать, являются ли изменения, внесенные инструментированием, безвредными, как и должно быть. Кроме того, результирующая оптимизация кода также затрагивается инструментированием, потому что функции регистрации информации о покрытии изменяют поток информации. Поэтому прежде чем заявлять зачет по структурному покрытию, необходимо повторно выполнить тесты без инструментирования и сравнить результаты с инструментированием и без него.

В-третьих, не все инструменты структурного покрытия полностью соответствуют критериям DO-178C. В частности, следует осторожно относиться к инструментам, используемым для измерения покрытия в параллельном коде, если они не разрабатывались для измерения покрытия при наличии конструкций многозадачности. Прежде чем вкладывать в инструмент время и деньги, его следует внимательно оценить и хорошо понять. Исследовательский отчет FAA под названием *Software Verification Tools Assessment Study* содержит на этот счет любопытные наблюдения. В рамках спонсированного FAA исследования был предложен тестовый набор для оценки коммерчески доступных инструментов структурного покрытия. Во всех трех оцененных инструментах были обнаружены аномалии. Это исследование показывает, что при выборе инструмента нужно соблюдать осторожность, поскольку некоторые инструменты могут не соответствовать определению структурного покрытия в DO-178C [17].

Наконец, результаты анализа структурного покрытия обычно либо включаются в результа-

ты верификации ПО, либо в этом документе на них дается ссылка. Любые пробелы структурного покрытия и их анализ будут внимательно рассматриваться сертифицирующими органами и/или их представителями, поэтому они должны быть сформулированы ясно и обоснованы тщательно.

9.8 Сообщения о проблемах

Хотя сообщения о проблемах в DO-178C относятся к управлению конфигурацией и подробнее рассматриваются в главе 10 этой книги, я обсуждаю их и в главе о верификации, потому что многие проблемы ПО обнаруживаются именно во время верификации. Сообщения о проблемах могут появляться на любой стадии проекта, но обычно их заводят после того, как элемент данных жизненного цикла, например требования, проект ПО, код, тестовые примеры или тестовые процедуры, был рассмотрен и включен в базовую конфигурацию. Сообщения о проблемах используют, чтобы управлять изменениями между базовыми версиями и работать с известными дефектами в программном обеспечении и данных его жизненного цикла.

В процессе верификации обычно выявляются проблемы в системных требованиях, требованиях к программному обеспечению, проекте ПО и исходном коде. Кроме того, могут быть обнаружены проблемы в планах, процессах жизненного цикла, инструментах или даже в аппаратуре. Сообщения о проблемах используются для документирования проблем процессов, данных и продукта. Как правило, сообщение о проблеме заводят только после того, как элемент данных жизненного цикла был рассмотрен, при необходимости обновлен и включен в базовую конфигурацию. После этого все изменения обычно оформляются через сообщение о проблеме. Некоторые компании используют два класса сообщений: один для кода, другой для ошибок в процессах и документации. Иногда существует и отдельный класс для будущих улучшений продукта. Лично я предпочитаю единую систему сообщений о проблемах для фиксации любых проблем или возможных изменений в ПО, но с хорошей схемой классификации для каждого сообщения. Например, сообщение можно отнести к проблемам кода, документации, процессов, будущим улучшениям и так далее.

Каждое сообщение о проблеме (problem report, PR) обычно содержит следующую информацию:

- номер сообщения о проблеме;
- дата составления сообщения о проблеме;
- краткое описание проблемы;
- затронутые элементы данных жизненного цикла и их версии;
- тип проблемы, например: код, проект ПО, требования, документация, улучшение, процесс, инструмент, иное;

- серьезность проблемы с точки зрения безопасности воздушного судна и соответствия требованиям, например: влияние на безопасность, влияние на сертификацию/соответствие, значительное функциональное/эксплуатационное влияние, незначительное функциональное/эксплуатационное влияние, только документация;
- приоритет проблемы с точки зрения разработчика программного обеспечения, например: исправить немедленно, исправить как можно скорее, исправить до следующего выпуска, исправить до сертификации, исправить по возможности, не исправлять;
- имя автора сообщения о проблеме и дата обнаружения проблемы;
- способ обнаружения проблемы;
- функциональная область проблемы;
- описание проблемы и способ ее воспроизведения;
- предлагаемое исправление, обычно это необязательное поле;
- подтверждающие данные, которые обычно прикладываются к сообщению о проблеме;
- лицо, назначенное для расследования проблемы;
- анализ проблемы и комментарии, то есть место, где документируется расследование;
- статус, например: открыт, в работе, исправлен, но еще не верифицирован, идет верификация, исправлен и верифицирован, не удастся воспроизвести, отложен, дубликат или отменен, если сообщение о проблеме оказалось ошибочным.

Следует отметить, что классификации по серьезности и приоритету не всегда используются одновременно. В некоторых системах регистрации сообщений о проблемах используется только одна классификация, в некоторых две. Это зависит от характера проекта. Схема классификации сообщений о проблемах должна быть описана в плане управления конфигурацией ПО (Software Configuration Management Plan, SCMP), чтобы ее применяли последовательно и она была понятна всем участникам. В главе 10 приведены дополнительные сведения о сообщениях о проблемах, включая некоторые рекомендуемые сертифицирующими органами схемы классификации.

Многим инженерам непросто писать хорошие сообщения о проблемах. Однако если делать это плохо, становится трудно подтвердить, что изменения надлежащим образом оценены, реализованы и верифицированы. Ниже приведены практические рекомендации по подготовке и сопровождению таких сообщений.

Рекомендация 1: Документируйте все проблемы в сообщениях о проблемах, за исключением случаев, когда документ, данные или продукт еще не прошли рассмотрение; в этом случае проблему следует зафиксировать в записи о рассмотрении. Устные сообщения и письма по электронной почте не отслеживаются достаточно надежно и не годятся для управляемой работы.

Рекомендация 2: Делайте краткое описание проблемы содержательным, лаконичным, точным и уникальным. Если проблема будет отложена, ее краткое описание может попасть в итоговый отчет разработки ПО, а значит, получит высокую видимость. Даже не отложенные сообщения о проблемах могут читать руководители, заказчики, системные инженеры, специалисты по безопасности, специалисты по гарантии качества ПО (software quality assurance, SQA), сотрудники по взаимодействию с сертифицирующим органом, а иногда и сами сертифицирующие органы. Поэтому краткое описание должно быть сформулировано так, чтобы оно было понятно даже человеку, не знающему деталей программного обеспечения. При этом оно должно быть полезно и тем, кто детали знает. Иными словами, на хорошее краткое описание стоит потратить дополнительное время, потому что жить ему, возможно, придется долго.

Рекомендация 3: Нумеруйте и классифицируйте каждое сообщение о проблеме. В ходе расследования классификация может измениться, но важно изначально иметь хотя бы общее представление о серьезности или приоритете проблемы. Более подробно подход к классификации обсуждается в главе 10.

Рекомендация 4: Убедитесь, что процесс классификации сообщений о проблемах четко задокументирован в плане управления конфигурацией ПО. Как уже отмечалось, в некоторых проектах будет несколько схем классификации, например одна для серьезности и одна для приоритета. Подход к классификации должен быть понятен разработчикам, заказчику, совету по управлению изменениями (change control board, CCB), сотрудникам по взаимодействию с сертифицирующим органом, руководству и всем прочим, кто читает сообщения о проблемах и принимает решения на их основе.

Рекомендация 5: Указывайте в каждом сообщении только одну проблему. Иногда возникает соблазн сгруппировать связанные проблемы вместе; однако закрывать такие сообщения неудобно, если часть проблем исправлена, а часть нет.

Рекомендация 6: Следите, чтобы сообщение о проблеме было читаемым и понятным. Нужны ясное и полное описание проблемы, а также конкретные сведения о том, как она была замечена, как ее воспроизвести и в чем именно она состоит. Очень помогают вложения с фрагментами проблемного элемента данных или снимками экрана. Если проблема описана неясно, надлежащих действий могут и не предпринять.

Рекомендация 7: Документируйте проблемы сразу после обнаружения. Даже очевидные ошибки нужно оформлять в виде сообщения, иначе их могут так и не исправить.

Рекомендация 8: Документируйте в сообщениях о проблемах и невозпроизводимые ошибки. По мере развития проекта такие ошибки могут проявиться отчетливее.

Рекомендация 9: Как правило, автор сообщения о проблеме должен тратить силы на объяснение проблемы, а не на ее решение. В сообщение можно включить некоторые предложения по исправлению, но сами решения рождаются уже в ходе расследования.

Рекомендация 10: Следите, чтобы сообщения о проблемах были профессиональными и безоценочными. Нет никакой пользы в том, чтобы обвинять программиста, руководство или архитектора. Поиск виноватых непродуктивен. Сообщение должно быть сосредоточено на технической проблеме, а не на людях.

Рекомендация 11: После того как о проблеме сообщили, ее должны рассмотреть один или несколько человек, чтобы определить дальнейшие шаги. Для оценки новых проблем и назначения ответственных обычно используют совет по управлению изменениями или аналогичную комиссию по рассмотрению сообщений о проблемах. Когда собраны все необходимые данные и подготовлена рекомендация по решению, такой совет принимает решение: одобрить решение или отложить его.

Рекомендация 12: Во время расследования инженер должен рассматривать и связанные вопросы. Заявленная проблема может быть лишь симптомом более крупной неприятности. Инженер, ведущий расследование, ищет первопричину, а не только симптомы.

Рекомендация 13: Помните, что расследование проблемы и отладка могут занять время. Некоторые проблемы очевидны, но на оценку других могут уйти недели. Расследования следует поручать опытным инженерам, которые хорошо понимают систему в целом и процесс разработки.

Рекомендация 14: Следите, чтобы предлагаемые решения ясно указывали, какие элементы будут изменены и затронуты, и какое ожидается изменение для каждого элемента. По сути, для каждого изменения нужен упрощенный анализ влияния. Чем лучше этот анализ задокументирован, тем легче потом поддержать сводный анализ влияния изменений для базовой версии ПО.

Рекомендация 15: После того как элемент данных был рассмотрен и включен в базовую конфигурацию, изменять его можно только через сообщение о проблеме или эквивалентный механизм санкционирования изменения. Это относится к данным категории контроля КК1 (control category 1), таким как требования, проект ПО, код, тестовые примеры верификации и процедуры. О категориях контроля КК1 и КК2 рассказывается в главе 10. При исправлении документированной проблемы возникает соблазн попутно поправить и очевидные ошибки. Само по себе это не неверно, но проблемы и решения должны быть задокументированы в сообщении о проблеме, а не исправлены молча, без следа.

Рекомендация 16: Рассматривайте проблемы в совокупности. На протяжении проекта раз-

работчики и руководство, а также заказчики, специалисты по качеству и сотрудники по взаимодействию с сертифицирующим органом, должны понимать общую картину по всему набору сообщений о проблемах.

Иногда проблемы связаны между собой, но если технические специалисты не чувствуют общей картины, они могут не заметить этих связей.

Рекомендация 17: Помните о риске “расползания” расследования ошибки (bug morphing) [9]. Это происходит, когда ошибка зарегистрирована, а инженер, ведущий расследование, уходит в сторону, например находит какие-то другие проблемы. Если в ходе расследования одного сообщения о проблеме обнаруживаются дополнительные проблемы, для них следует заводить отдельные сообщения о проблемах.

Рекомендация 18: Не бойтесь сообщать о возможных дубликатах. Некоторые разработчики не хотят заводить сообщение о проблеме, потому что боятся, что оно окажется дубликатом. Им стоит быстро рассмотреть список уже существующих сообщений о проблемах и отправить свое сообщение, если они не видят, что проблема явно уже зафиксирована. Стоит также отметить, что в проблемной области программного обеспечения может существовать несколько сообщений о проблемах, которые связаны между собой, но немного различаются, потому что у одной глубинной проблемы часто бывает несколько симптомов. Поэтому лучше завести сообщение о проблеме, которое, возможно, окажется дубликатом, чем вообще не сообщить о проблеме. Более тщательную оценку на предмет дублирования выполнит совет по управлению изменениями или комиссия по рассмотрению сообщений о проблемах.

Рекомендация 19: Чтобы обеспечить качество и единообразие сообщений о проблемах, назначьте инженера для рассмотрения этих сообщений с точки зрения читаемости, точности, полноты и тому подобного. Нередко этот человек также руководит советом по управлению изменениями или комиссией по рассмотрению сообщений о проблемах и следит за тем, чтобы сообщение было достаточно проработано для рассмотрения комиссией.

Рекомендация 20: Перед закрытием сообщения о проблеме проверьте реализацию решения или решений. Процесс верификации может включать повторное рассмотрение требований, проекта ПО, кода и тестовых данных. Обычно предпочтителен независимый рецензент. Элемент данных должен пройти тот же уровень строгости, что и изначально, то есть через те же процессы, контрольные перечни и так далее.

Рекомендация 21: Если сообщение о проблеме откладывается, внесите в него обоснование отсрочки. Это обоснование должно объяснять, почему, если проблему не исправить до сертификации, не возникает проблем с безопасностью, эксплуатацией, характеристиками или соответствием требованиям. Такое обоснование будет рассматриваться очень внимательно, поэтому оно должно быть сформулировано ясно и хорошо подкреплено данными. Обос-

нование оценивают совет по управлению изменениями и/или комиссия по рассмотрению сообщений о проблемах, заказчик, специалисты по безопасности и сотрудники по взаимодействию с сертифицирующим органом, чтобы решить, допустима ли отсрочка. После согласования решения об отсрочке это обоснование включают в итоговый отчет разработки ПО для согласования с сертифицирующими органами и/или их представителями.

Рекомендация 22: Подключайте гарантию качества ПО к процессу отсрочки и закрытия сообщений о проблемах. Служба гарантии качества ПО может быть членом совета по управлению изменениями или иметь собственную процедуру одобрения.

9.9 Рекомендации по процессам верификации

Некоторые утверждают, что применительно к верификации не существует такого понятия, как *наилучшая практика* (best practice), поскольку слишком многое зависит от конкретной ситуации. Однако есть ряд практик, которые делают ход проекта более гладким и повышают вероятность успеха. В этом разделе приводится сводка рекомендаций, основанных на моем участии во множестве проектов. Часть из них суммирует материал, рассмотренный выше, а часть еще не обсуждалась.

Рекомендация 1: Планируйте рано и часто корректируйте план. Ни один проект не способен предсказать все проблемы, которые возникнут. Тем не менее планирование важно и может уменьшить хотя бы часть крупных проблем. Но помимо самого планирования необходимо постоянно вносить корректировки, чтобы учитывать реальные проблемы.

Рекомендация 2: Стремитесь к реалистичному графику. Выдуманнные графики – один из моих личных раздражителей. Планирование, основанное на мечтах, и управление с головой в песке просто не работают. Однажды я работала над графиком и сказала руководителю, что мне нужно сначала проанализировать задачи, прежде чем я смогу спрогнозировать сроки. Он посмотрел на меня так, будто я сошла с ума. Он сказал: “На мне сейчас менеджерская шляпа”. Я не понимаю, почему *менеджерская шляпа* не может опираться на реальность. Как любит говорить один мой коллега: “Хорошие руководители не игнорируют реальность”. Слишком часто планирование строится на деловой повестке, а не на реальности. Большинство инженеров будут работать как одержимые, чтобы уложиться в реалистичный, пусть и напряженный график. Но слишком большое число невозможных или фантазийных сроков приводит к деморализации и потере мотивации.

Рекомендация 3: Проектируйте с расчетом на робастность. Почти невозможно добиться робастности одним только тестированием. Гораздо лучше заранее предусматривать аномальные входные данные и проектировать систему так, чтобы она справлялась с ними.

Рекомендация 4: Проектируйте с расчетом на пригодность к тестированию. Как отме-

чалось ранее, полезно, чтобы разработчики строили систему так, чтобы она поддерживала тестирование. Например, если некоторые структуры данных видимы на более внешнем уровне, проводить тестирование проще, чем верифицировать поведение по данным, до которых трудно добраться. Полезно привлекать специалиста по тестированию к рассмотрению требований и проекта ПО, потому что этот специалист будет заранее думать о тестовых аспектах.

Рекомендация 5: Начинайте верификацию рано и ведите ее на протяжении всего проекта. Верификация должна начинаться как можно раньше в ходе разработки. Рассмотрения требований, проекта ПО и кода должны выполняться по мере появления данных. Исключительно полезно проводить неформальные технические рассмотрения данных до формального рассмотрения. Откладывать верификацию на конец проекта неэффективно, может оказаться очень дорого и способно повредить отношениям как с сертифицирующим органом, так и с заказчиком.

Рекомендация 6: Определяйте роли во время рассмотрений. Во время взаимных рассмотрений полезно распределять роли. Например, один человек может сосредоточиться на том, насколько хорошо тесты отрабатывают требования; один – на точности трассировок; один – на корректности и точности тестовых примеров; а еще один – выполнять тестовые процедуры, чтобы определить их повторяемость.

Рекомендация 7: Привлекайте специалистов по тестированию уже на этапе рассмотрений. Если они участвуют в рассмотрении требований и проекта ПО, они могут помочь убедиться, что программное обеспечение пригодно для тестирования, и одновременно познакомиться с требованиями, которые им предстоит проверять.

Рекомендация 8: Начинайте работы по тестированию рано. По мере созревания требований кто-то должен начать планирование тестирования и разработку тестовых примеров.

Рекомендация 9: Дайте специалистам по тестированию время и возможность понять систему. Чем лучше они понимают назначение системы и ее фактическую функциональность, тем лучше они смогут выявлять ошибки. Когда требования распределены между специалистами по тестированию и ни у кого нет целостного понимания системы, тестирование оказывается менее эффективным.

Рекомендация 10: Знайте среду. Специалисты по тестированию должны понимать эксплуатационную среду программного обеспечения. Для эффективного тестирования также важно знание среды разработки, аппаратной части и интерфейсов.

Рекомендация 11: Специалистов по тестированию следует поощрять и им следует позволять ставить под сомнение все. Очевидно, что вопросы не обязательно произносить вслух,

но лучший способ найти ошибки – постоянно задаваться вопросами “а что, если?”, “как это работает?” или “почему это работает?” Нелюбопытный верификатор, как правило, неэффективен.

Рекомендация 12: Используйте опытных людей и позволяйте им применять свой опыт. Очевидно, что круг экспертов ограничен. Тем не менее опытных специалистов по тестированию следует использовать на ключевых задачах. Младшие инженеры тоже могут играть свою роль, и часть их работы бывает поразительно изобретательной. Но старшим и более опытным сотрудникам следует позволять вести за собой. Некоторые компании вводят форму *парного тестирования* (pair testing), когда два человека работают над тестированием вместе. Это может быть эффективным способом обучать младшего инженера и использовать опытных специалистов по тестированию.

Рекомендация 13: Поощряйте специалистов по тестированию думать шире самих требований, особенно на ранних этапах проекта. Если тестируются только требования, а сами требования неверны или неполны, серьезные проблемы могут быть выявлены поздно или не быть выявлены вообще.

Рекомендация 14: Сначала тестируйте критичную и ключевую функциональность. Если базовые вещи не работают, все остальное уже не будет иметь большого значения.

Рекомендация 15: Если что-то сбивает с толку, это следует тестировать больше. Если требования, проект ПО и/или код сбивают с толку, обычно на то есть причина. Это может быть связано либо с тем, что разработчик недостаточно хорошо понял проблему, что приводит к пропускам или ошибкам, либо с тем, что разработчик понял ее слишком хорошо, что ведет к чрезмерному упрощению. Если требования или проект ПО сбивают с толку, это часто приводит к ошибкам в коде. Во многих случаях такой запутанный участок нужно перепроектировать и перекодировать или хотя бы привести в порядок.

Рекомендация 16: Ожидайте, что на проверку сложных областей уйдет больше времени. Как отмечалось ранее, ошибки обычно прячутся в пещерах сложности. Если что-то сложно, скорее всего там будет больше проблем, и на верификацию и исправление выявленных дефектов уйдет больше времени.

Рекомендация 17: Осознайте, что качество нужно встраивать, а не пытаться обеспечить его только на этапе тестирования. Тестирование является индикатором общего качества продукта, но ждать этапа тестирования, чтобы заняться качеством, уже слишком поздно. Качество нужно встраивать проактивно. Привлечение специалистов по тестированию к процессам разработки, а также ранняя разработка и выполнение тестов (1) помогают выявлять проблемы, пока их еще можно исправить с разумными затратами, и (2) повышают качество. Иногда тестирование выявляет проблему, которая приводит к перепроектирова-

нию и повторной реализации; такие ошибки лучше находить как можно раньше.

Рекомендация 18: Специалисты по тестированию должны сосредоточиваться на технической функциональности, а не на людях. Как уже упоминалось, специалистов по тестированию часто воспринимают как негативную группу. Их негативность помогает выявлять проблемы, которые нужно решить. Однако им нужно следить за тем, чтобы направлять эту негативную энергию на программное обеспечение, а не на людей, которые его написали.

Рекомендация 19: Используйте как можно больше независимой верификации. Хотя DO-178C не требует независимости для многих мероприятий по верификации, определенная степень независимости весьма эффективна. Находить ошибки в собственной работе действительно трудно.

Рекомендация 20: Пишите точные сообщения о проблемах и следите за общим состоянием проблем проекта. Сообщения о проблемах следует создавать сразу же, как только замечены проблемы. Точно так же сами сообщения должны быть максимально точными: плохие данные никому не помогают. Кроме того, руководству следует регулярно читать сообщения о проблемах, чтобы надлежащим образом управлять проектом.

Рекомендация 21: Автоматизируйте с умом. Автоматизацию следует применять только тогда, когда в этом есть смысл. Некоторые проекты автоматизируют просто ради самой автоматизации. Инженеры любят создавать инструменты. Иногда на создание и квалификацию инструмента может уйти больше времени, чем на ручную проверку продукта. Кроме того, инструменты не заменяют необходимость думать. Если использовать их без должной осмотрительности, они могут создавать ложное чувство уверенности.

Рекомендация 22: Создайте команде условия для успеха. Эффективный руководитель развивает командную работу, побуждает каждого участника делать все возможное, использует разнообразие подходов, поддерживает невраждебную и продуктивную рабочую среду, поощряет талант, побуждает специалистов по тестированию находить ошибки, проактивно занимается проблемами и принимает решения на основе данных, а не ощущений, особенно когда речь идет о графиках.

Рекомендация 23: Развивайте творческий подход. Тщательное тестирование требует творческого подхода. Иногда тестирование требует даже большей изобретательности, чем разработка, потому что специалистам по тестированию нужно думать о том, как сломать программное обеспечение, а не о том, как заставить его работать. Творческий подход можно поощрять, предоставляя свободное время на размышления, которое я называю *временем для идей* (dream time), а также устраивая интересные занятия и соревнования.

Рекомендация 24: Инвестируйте в обучение специалистов по тестированию. Хорошие спе-

специалисты по тестированию постоянно учатся. Руководителям следует создавать возможности для обучения и роста, даже если на это уходит день или два, которые иначе были бы потрачены на проект.

Рекомендация 25: Внедряйте непрерывное совершенствование. Однажды я видела комикс, где человек рубил деревья в лесу топором. У него были сроки, и у него не было времени учиться пользоваться бензопилой, лежавшей в его грузовике. Иногда мы настолько сосредоточены на текущей задаче, что упускаем возможности для улучшения. Разумеется, в конце проекта нужно зафиксировать извлеченные уроки и предпринять действия. Но не менее ценно корректировать курс и по ходу проекта, когда что-то не работает или когда есть более эффективный способ выполнить работу.

Рекомендация 26: Выявляйте риски и проблемы и проактивно работайте с ними. По мере появления проблем ими нужно заниматься. Сами по себе они не исчезнут.

Рекомендация 27: Если тестирование передается на аутсорсинг полностью или частично, не просто “перекидывайте его через стену”. Аутсорсинг или субподряд потребуют плотного надзора, обучения и постоянной коммуникации. Подробнее это обсуждается в главе 26.

Рекомендация 28: Ведите основной список типовых проблем и ошибок и следите, чтобы команда его знала. Полезно составить список типовых проблем на основе опыта опытных специалистов по тестированию, сообщений о проблемах и литературы. Со временем этот список можно обновлять. Специалистов по тестированию следует знакомить с типичными ошибками. Разумеется, они не должны ограничивать верификацию только этим списком, но он станет хорошей отправной точкой для поиска проблем в системе.

Рекомендация 29: Будьте готовы к трудностям. Никогда не знаешь, с какими именно трудностями столкнешься при верификации программного обеспечения. Нужно быть готовым адаптироваться и держать удар. Одни из самых распространенных трудностей таковы: неоднозначные требования, отсутствующие требования, определение того, какой уровень робастности достаточен, работа под давлением графика, поскольку тестирование обычно является последним мероприятием, отработка архитектуры наряду с требованиями, управление изменениями, то есть отслеживание изменений требований, проекта ПО и кода, поддержание морального духа, работа с проблемами, обнаруженными на поздних этапах процесса, и поддержание адекватного соотношения между числом специалистов по тестированию и разработчиков. У каждого проекта есть собственный особый набор проблем. Будьте готовы ко всему.

Ссылки

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Вашингтон, округ Колумбия: RTCA, Inc., декабрь 2011).
2. Certification Authorities Software Team (CAST), Verification independence, Position Paper CAST-26 (январь 2006, ред. 0).
3. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Вашингтон, округ Колумбия: RTCA, Inc., декабрь 2011).
4. RTCA DO-254, *Design Assurance Guidance for Airborne Electronic Hardware* (Вашингтон, округ Колумбия: RTCA, Inc., апрель 2000).
5. Certification Authorities Software Team (CAST), Addressing cache in airborne systems and equipment, Position Paper CAST-20 (июнь 2003, ред. 1).
6. С. Kaner, J. Bach и В. Pettichord, *Lessons Learned in Software Testing* (Нью-Йорк: John Wiley & Sons, 2002).
7. G. J. Myers, *The Art of Software Testing* (Нью-Йорк: John Wiley & Sons, 1979).
8. E. Kit, *Software Testing in the Real World* (Харлоу, Англия, Великобритания: ACM Press, 1995).
9. А. Page, К. Johnston и В. Rollison, *How We Test Software at Microsoft* (Редмонд, штат Вашингтон: Microsoft Press, 2009).
10. С. Kaner, J. Falk и Н. Q. Nguyen, *Testing Computer Software*, 2-е изд. (Нью-Йорк: John Wiley & Sons, 1999).
11. J. A. Whittaker, *How to Break Software: A Practical Guide to Testing* (Бостон, штат Массачусетс: Addison-Wesley, 2003).
12. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7-е изд. (Нью-Йорк: McGraw-Hill, 2010).
13. К. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski и L. K. Rierson, *A Practical Tutorial on Modified Condition/Decision Coverage*, NASA/TM2001-210876 (Хэмптон, штат Вирджиния: Langley Research Center, май 2001).
14. Certification Authorities Software Team (CAST), Structural coverage of object code, Position Paper CAST-17 (июнь 2003, ред. 3).
15. Certification Authorities Software Team (CAST), Guidelines for approving source code to object code traceability, Position Paper CAST-12 (декабрь 2002).

16. Certification Authorities Software Team (CAST), Clarification of structural coverage analyses of data coupling and control coupling, Position Paper CAST-19 (январь 2004, ред. 2).
17. V. Santhanam, J. J. Chilenski, R. Waldrop, T. Leavitt и K. J. Hayhurst, *Software Verification Tools Assessment Study*, DOT/FAA/AR-06/54 (Вашингтон, округ Колумбия: Office of Aviation Research, июнь 2007).

Рекомендуемое чтение

1. K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski и L. K. Rierison, *A Practical Tutorial on Modified Condition/Decision Coverage*, NASA/TM2001-210876 (Хэмптон, штат Вирджиния: Langley Research Center, май 2001). В этом учебном материале предложен практический подход к оценке авиационного программного обеспечения на соответствие цели DO-178B (а теперь и DO-178C) по MC/DC (таблица A-7 DO-178C, цель 5). В материале представлен пятишаговый подход к оценке покрытия MC/DC без инструмента покрытия. Также рассматриваются факторы, которые следует учитывать при выборе и/или квалификации инструмента анализа структурного покрытия. Кроме того, обсуждаются советы по рассмотрению артефактов MC/DC и типичные подводные камни анализа структурного покрытия.
2. J. J. Chilenski, *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*, DOT/FAA/AR-01/18 (Вашингтон, округ Колумбия: Office of Aviation Research, апрель 2001). В этом отчете сравниваются три формы MC/DC и приводится обоснование того, почему MC/DC следует включать в процесс разработки программных систем. Эти три формы MC/DC сравниваются теоретически и эмпирически с точки зрения минимальной вероятности обнаружения ошибки и простоты достижения.
3. V. Santhanam, J. J. Chilenski, R. Waldrop, T. Leavitt и K. J. Hayhurst, *Software Verification Tools Assessment Study*, DOT/FAA/AR-06/54 (Вашингтон, округ Колумбия: Office of Aviation Research, июнь 2007). В этом отчете документировано исследование критериев, позволяющих эффективно оценивать инструменты анализа структурного покрытия для проектов, нацеленных на соответствие DO-178B (а теперь и DO-178C). В рамках исследования был предложен набор тестов для повышения объективности и единообразия при применении критериев квалификации инструмента структурного покрытия. Прототипный набор тестов выявил аномалии в каждом из трех оцененных инструментов анализа покрытия, продемонстрировав, что набор тестов действительно может помочь при оценке совместимости инструмента с целями DO-178B (а теперь и DO-178C).
4. J. J. Chilenski и J. L. Kurtz, *Object-Oriented Technology Verification Phase 3 Handbook*:

Structural Coverage at the Source-Code and Object-Code Levels, DOT/FAA/AR-07/17 (Вашингтон, округ Колумбия: Office of Aviation Research, июнь 2007). В этом руководстве приведены указания по выполнению целей анализа структурного покрытия DO-178B (а теперь и DO-178C) на уровне исходного кода по сравнению с уровнем объектного кода или исполняемого объектного кода при использовании объектно-ориентированной технологии (ООТ) в коммерческой авиации. Выявлены различия между анализом покрытия исходного кода и объектного кода или исполняемого объектного кода для объектно-ориентированных возможностей и MC/DC. Для каждой выявленной проблемы предложен подход к работе с этими различиями. Хотя основной акцент сделан на объектно-ориентированной технологии (ООТ), многие из изложенных идей применимы и к проектам без ООТ.

5. CAST-17, *Structural coverage of object code* (ред. 3, июнь 2003). В этом документе, подготовленном международной группой сертифицирующих органов Certification Authorities Software Team (CAST), объясняются некоторые мотивы анализа структурного покрытия на уровне объектного кода или исполняемого объектного кода и перечисляются вопросы, которые необходимо учитывать при использовании такого подхода.

6. J. J. Chilenski и J. L. Kurtz, *Object-Oriented Technology Verification Phase 2 Handbook: Data Coupling and Control Coupling*, DOT/FAA/AR07/19 (Вашингтон, округ Колумбия: Office of Aviation Research, август 2007). В этом руководстве приведены указания по верификации межкомпонентной связности по данным и по управлению внутри объектно-ориентированной технологии (ООТ) в коммерческой авиации. Покрытие межкомпонентных зависимостей рассматривается как допустимая мера интеграционного тестирования как для программного обеспечения без ООТ, так и с ООТ, для выполнения цели 8 таблицы А-7 DO-178B (а теперь и DO-178C). Такой подход известен как интеграционное тестирование на основе межкомпонентной связности (coupling-based integration testing).

*Подробности о целях верификации обсуждаются далее; пока акцент сделан на том, какие именно цели требуют независимости.

†CAST – это группа международных сертифицирующих органов, которая стремится согласовывать свои позиции по вопросам бортового программного обеспечения и авиационной электронной аппаратуры в документах CAST.

*Разделы 6.3.4.f и 6.3.5 DO-178C указывают на некоторые из этих аспектов как на часть верификации кода и интеграции.

*При сборе данных о времени выполнения в наихудшем случае (WCET, Worst-Case

Execution Time) нужно соблюдать осторожность. Вполне возможно получить значение WCET больше 100% и при этом сохранить безопасную работу системы. Например, ветви с наихудшим временем выполнения в одной процедуре могут быть взаимоисключающими. Проблема усугубляется, когда взаимоисключение возникает между компонентами, то есть наихудшее время в компоненте А не может возникнуть одновременно с наихудшим временем в компоненте В. Но если оба компонента находятся в одном исполнительном кадре, данные WCET могут суммировать оба значения.

†CAST-20, озаглавленный *Addressing cache in airborne systems and equipment* [5], содержит дополнительные сведения о проблемах времени выполнения в наихудшем случае (WCET) при использовании кэша и/или конвейерной обработки.

*Системы интегрированной модульной авионики (ИМА) кратко обсуждаются в главах 20 и 21. Аналогично, данные конфигурации рассматриваются в главе 22.

*NaN может быть quiet NaN (тихим NaN) или signaling NaN (сигнальным NaN).

†Следует отметить, что некоторые технологии, например формальные методы, могут уменьшать необходимость в части мероприятий по тестированию. Формальные методы обсуждаются в главе 16.

*Согласно разделу 6.4.2 DO-178C.

*Вместо одного живого документа в некоторых проектах имеется план тестирования, который завершается заранее, а также документ с верификационными вариантами и процедурами, который развивается на протяжении всего проекта.

*Возможность писать тесты по модулям на уровне требований низкого уровня будет зависеть от того, как организованы требования низкого уровня.

*Инспекции кода особенно субъективны, когда используются для подтверждения того, что исполняемый объектный код удовлетворяет требованиям. Когда они применяются для выполнения целей таблицы А-6 DO-178C, акцент инспекции кода делается на том, как именно исполняемый объектный код удовлетворяет требованиям, а не на повторении рассмотрения кода, выполняемого для целей таблицы А-5 DO-178C.

*Раздел 4.4.3.b DO-178C.

†Раздел 6.4.1.a DO-178C.

‡Глава 16 FAA Order 8110.49 (изменение 1) содержит некоторые указания по управлению средами разработки и верификации при использовании эмулятора или симулятора. Документ можно найти на сайте FAA: www.faa.gov

*Даже автоматизированные тесты требуют процедур и нуждаются в журнале тестирования.

**Redline* – это изменение процедуры. Обычно оно документируется красной ручкой на бумажной копии или с помощью отслеживания изменений в электронной копии.

*Мертвый код – это: “исполняемый объектный код (или данные), который существует в результате ошибки разработки программного обеспечения, но не может быть выполнен (код) или использован (данные) ни в одной эксплуатационной конфигурации среды целевого вычислителя. Он не трассируется на системное требование или требование к ПО. Следующие исключения часто ошибочно относят к мертвому коду, хотя они необходимы для реализации требований/проекта ПО: встроенные идентификаторы, конструкции защитного программирования для повышения робастности и отключенный код, такой как неиспользуемые библиотечные функции” [1].

†Посторонний код – это: “код (или данные), который не трассируется ни на какое системное требование или требование к ПО. Примером постороннего кода является унаследованный код, который ошибочно сохранили, хотя его требования и тестовые примеры были удалены. Другой пример постороннего кода – мертвый код” [1].

*CAST-12, озаглавленный *Guidelines for approving source code to object code traceability*, содержит дополнительные сведения по этой теме [15].

*Скобки добавлены для ясности.

Глава 10. Управление конфигурацией ПО

Сокращения

Сокращение	Расшифровка
KK1	Категория контроля 1 (control category 1)
KK2	Категория контроля 2 (control category 2)
CCB	Совет по управлению изменениями (change control board)
CD	Компакт-диск (compact disk)
CIA	Анализ воздействия изменений (change impact analysis)
CRC	Циклический избыточный код (cyclic redundancy check)
DVD	Цифровой видеодиск (digital video disk)
EASA	Агентство Европейского союза по авиационной безопасности (European Union Aviation Safety Agency)
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
PR	Сообщение о проблеме (problem report)
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
SAS	Итоговый отчёт разработки ПО (Software Accomplishment Summary)
SCI	Указатель конфигурации ПО (Software Configuration Index)
SCM	Управление конфигурацией ПО (software configuration management)
SCMP	План управления конфигурацией ПО (Software Configuration Management Plan)
SECI	Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index)
SQA	Гарантия качества ПО (software quality assurance)
SQAP	План гарантии качества ПО (Software Quality Assurance Plan)

10.1 Введение

10.1.1 Что такое управление конфигурацией ПО?

Управление конфигурацией ПО (software configuration management, SCM) – это интегральный процесс, который сопровождает весь жизненный цикл ПО от начала до конца. Он охватывает все области жизненного цикла ПО и влияет на все данные и процессы. Бэбич пишет:

«Управление конфигурацией – это искусство идентификации, организации и управления изменениями программного обеспечения, создаваемого командой программистов. Его цель состоит в максимизации продуктивности за счет минимизации ошибок» [1].

Управление конфигурацией ПО нужно не только для исходного кода, хотя так часто думают; оно требуется для всех данных жизненного цикла ПО. Все данные и документация, используемые для создания программного обеспечения, его верификации и подтверждения соответствия, требуют некоторого уровня управления конфигурацией. Иначе говоря, все данные жизненного цикла ПО, перечисленные в разделе 11 стандарта DO-178C (КТ-178С), требуют управления конфигурацией. Строгость применяемого процесса управления конфигурацией зависит от уровня ПО и характера артефакта. В DO-178C используется понятие КК1/КК2, то есть категории контроля 1 и 2, чтобы определить, какой объем конфигурационного управления применяется к элементу данных. К элементу данных, отнесенному к КК1, должны применяться все мероприятия управления конфигурацией, определенные в DO-178C. Для элемента данных категории КК2, напротив, может применяться только часть этих мероприятий. Вопрос КК1/КК2 рассматривается далее в этой главе.

В этой главе рассматриваются мероприятия управления конфигурацией, которые требуются DO-178C и считаются лучшими практиками в любой области, критически важной для безопасности. Особое внимание уделено процессам регистрации сообщений о проблемах, анализа воздействия изменений и управления средой, поскольку именно с этими процессами компании часто испытывают затруднения на практике.

10.1.2 Зачем нужно управление конфигурацией ПО?

Разработка программного обеспечения во всех областях, включая области, критически важные для безопасности, идет под сильным давлением. От инженеров-программистов требуется разрабатывать сложные системы в сжатые сроки и в рамках ограниченного бюджета. Ожидается, что они будут быстро вносить обновления и поддерживать высокое качество ПО, соответствующего стандартам и нормативным требованиям.

Чтобы выжить в этой довольно беспощадной конкурентной среде, организациям нужен механизм, который позволяет держать ситуацию под контролем. Иначе возникают хаос и путаница, а это может привести к провалу продукта или проекта и поставить компанию на грань ухода с рынка. Правильно внедренная система управления конфигурацией ПО как раз и является таким механизмом, который помогает командам разработки создавать высококачественное программное обеспечение без хаоса и путаницы [2].

Хорошее управление конфигурацией помогает предотвращать такие проблемы, как отсутствие модулей исходного кода, невозможность найти последнюю версию файла, повторное появление уже исправленных ошибок, отсутствие требований, невозможность опре-

делить, что именно изменилось и когда, перезапись результатов работы друг друга двумя разработчиками при обновлении общего файла, и многое другое. Управление конфигурацией уменьшает такие проблемы за счет координации работы нескольких людей, занятых в одном проекте. При надлежащем внедрении управление конфигурацией «предотвращает техническую анархию, позволяет избежать неловкой ситуации из-за неудовлетворенности заказчика и поддерживает согласованность между продуктом и информацией о продукте» [3].

Управление конфигурацией также обеспечивает эффективное управление изменениями. Системы, в которых активно используется программное обеспечение, будут меняться: такова природа ПО. Прессман пишет: «Если вы не управляете изменениями, они начинают управлять вами. А это никогда не приводит ни к чему хорошему. Очень легко допустить, чтобы поток неконтролируемых изменений превратил хорошо организованный программный проект в хаос» [4]. Поскольку изменения происходят очень часто, ими необходимо эффективно управлять. Хорошее управление конфигурацией дает возможность управлять изменениями за счет: (1) выявления элементов данных, которые, вероятно, будут меняться, (2) определения отношений между элементами данных, (3) определения подхода к управлению редакциями данных, (4) управления реализуемыми изменениями и (5) сообщения о внесенных изменениях.

Эффективное управление конфигурацией дает много преимуществ команде разработки ПО, организации в целом и заказчику, например:

- поддерживает целостность программного обеспечения за счет упорядоченных задач и мероприятий; укрепляет доверие сертифицирующих органов и заказчиков;
- способствует более высокому качеству и, следовательно, большей безопасности программного обеспечения; обеспечивает управление данными жизненного цикла, требуемыми DO-178C; гарантирует, что конфигурация ПО известна и корректна; помогает соблюдать сроки и бюджет проекта; позволяет формировать базовые версии ПО и элементов данных; позволяет отслеживать изменения базовых версий;
- предотвращает путаницу и улучшает взаимодействие между участниками команды разработки за счет систематического подхода к управлению данными; помогает избегать неожиданностей или хотя бы уменьшать их количество и сокращать потери времени;
- дает средства для выявления, регистрации и устранения проблем в коде и сопровождающих его данных жизненного цикла;
- помогает создать контролируемую среду для разработки, верификации, тестирования и воспроизведения программного обеспечения;

- гарантирует, что программное обеспечение можно будет повторно собрать даже спустя годы после первоначальной разработки; обеспечивает учет состояния на протяжении всего проекта;
- создает основу для принятия решений разработчиками и заказчиками; дает возможность воспроизводить проблемы при расследовании дефектов; создает основу для совершенствования процесса; предоставляет данные для подтверждения завершенности программного обеспечения; поддерживает долгосрочное сопровождение.

Риски отсутствия хорошего управления конфигурацией весьма велики. Плохое управление конфигурацией ПО приводит к потерям времени, денег, качества и доверия. Недавно мне довелось консультировать проект в компании со слабым управлением конфигурацией ПО. У них были сильные инженеры, но они не могли стабильно воспроизводить ни программное обеспечение, ни сам авиационный блок. Это привело к огромным задержкам, усиленному надзору со стороны Федерального управления гражданской авиации США (FAA, Federal Aviation Administration) и дорогостоящим штрафам из-за опоздания продукта.

Хотя читать про управление конфигурацией не особенно увлекательно, его важность трудно переоценить. Проекты становятся слишком «увлекательными» именно тогда, когда управление конфигурацией ПО не применяется. Обычно это не тот вид увлекательности, который кому-то нужен.

10.1.3 Кто отвечает за внедрение управления конфигурацией ПО?

За управление конфигурацией ПО отвечает каждый, кто участвует в процессе разработки программного обеспечения. Разработка ПО во всех областях, включая критичную по безопасности, ведется в условиях высокого давления. Всех разработчиков необходимо обучать преимуществам хорошего управления конфигурацией ПО, рискам плохого управления конфигурацией, лучшим практикам SCM и конкретным корпоративным процедурам. Такое обучение помогает инженерам в целом работать лучше. При правильном внедрении и автоматизации ежедневное выполнение процедур управления конфигурацией ПО не должно становиться для разработчиков сложной задачей.

В прошлом управление конфигурацией ПО было ручным и трудоемким процессом. Однако сегодня, благодаря наличию хороших инструментов управления конфигурацией ПО, этот процесс можно внедрить без чрезмерной нагрузки на разработчиков.

Но мысль о том, что инструменты управления конфигурацией ПО сами обо всем позаботятся, может стать прямым путем к катастрофе. Многие мероприятия управления конфигурацией ПО, включая управление изменениями, управление сборками и выпусками, а также конфигурационные аудиты, требуют вмешательства человека и его суждения. Инструменты управления конфигурацией ПО делают эти задачи проще, но заменить человеческий интел-

лект и принятие решений они не могут [2].

Управление конфигурацией ПО – ответственность всех участников. Для него нужны коммуникация и командная работа между теми, кому требуются программное обеспечение и данные, теми, кто их создает, и теми, кто ими пользуется. Хорошее управление конфигурацией ПО начинается и заканчивается коммуникацией. Ниже приведены несколько рекомендаций, которые помогают наладить коммуникацию, необходимую для эффективного управления конфигурацией ПО:

- обеспечить, чтобы цели и задачи были ясно сформулированы;
- убедиться, что все заинтересованные стороны понимают цели и задачи;
- обеспечить сотрудничество всех заинтересованных сторон и устранять любые проблемы, мешающие взаимодействию;
- обеспечить, чтобы все процессы были документированы и ясно понятны всем заинтересованным сторонам;
- регулярно предоставлять обратную связь;
- обеспечить доступность данных, необходимых для принятия решений; решать возникающие проблемы по мере их появления.

10.1.4 Что включает в себя управление конфигурацией ПО?

Помимо разработки плана управления конфигурацией ПО (Software Configuration Management Plan, SCMP), который является выходным артефактом целей таблицы A-1 стандарта DO-178C, стандарт DO-178C в таблице A-8 определяет шесть целей процесса управления конфигурацией. Каждая цель обязательна для всех уровней ПО. Эти шесть целей таковы [5]:

- *DO-178C, таблица A-8, цель 1:* «Конфигурационные элементы идентифицированы».
- *DO-178C, таблица A-8, цель 2:* «Базовые версии и трассируемость установлены».
- *DO-178C, таблица A-8, цель 3:* «Установлены сообщения о проблемах, управление изменениями, рассмотрение изменений и учет состояния конфигурации».
- *DO-178C, таблица A-8, цель 4:* «Установлены архивирование, извлечение и выпуск».
- *DO-178C, таблица A-8, цель 5:* «Установлено управление загрузкой программного обеспечения».
- *DO-178C, таблица A-8, цель 6:* «Установлено управление средой жизненного цикла ПО».

Эти цели применяются для обеспечения целостности, подотчетности, воспроизводимости,

прозрачности, координации и контроля данных жизненного цикла ПО по мере их изменения. Для выполнения целей DO-178C требуется несколько мероприятий, включая идентификацию конфигурации, установление базовых версий, трассируемость, сообщения о проблемах, управление изменениями, рассмотрение изменений, учет состояния конфигурации, выпуск, архивирование и извлечение, управление загрузкой и управление средой. Каждое из этих мероприятий рассматривается в следующем разделе.

10.2 Мероприятия по управлению конфигурацией ПО

В этом разделе описываются мероприятия процесса управления конфигурацией ПО. Каждое мероприятие должно быть подробно определено в плане управления конфигурацией ПО на раннем этапе проекта. Этот план помогает команде управления конфигурацией ПО и проектной команде понимать процедуры, роли и обязанности, которые им предстоит выполнять в ходе проекта для поддержки управления конфигурацией ПО. И команда управления конфигурацией ПО, и проектная команда должны пройти обучение по ожиданиям и требованиям управления конфигурацией. План служит основой для обучения команд и внедрения нужных процессов управления конфигурацией. Дополнительные сведения о плане управления конфигурацией ПО приведены в главе 5.

10.2.1 Идентификация конфигурации

«Идентификация конфигурации является одним из краеугольных камней управления конфигурацией, поскольку невозможно управлять тем, что не идентифицировано» [6]. Это первое мероприятие в управлении конфигурацией. Идентификация конфигурации «определяет элементы, подлежащие управлению, устанавливает схемы идентификации элементов и их версий, а также определяет инструменты и методы, которые будут использоваться для получения и управления контролируемыми элементами» [2]. Идентификация конфигурации задает отправную точку для остальных мероприятий управления конфигурацией ПО; это первая крупная функция SCM, которую необходимо запустить в проекте. По сути, она является предварительным условием для других мероприятий управления конфигурацией ПО, поскольку все остальные мероприятия используют результаты идентификации конфигурации [2].

Идентификация конфигурации обеспечивает возможность: (1) идентифицировать компоненты системы на протяжении всего их жизненного цикла и (2) устанавливать трассируемость между программным обеспечением и его данными жизненного цикла. Каждая единица конфигурации должна быть однозначно идентифицирована. Метод идентификации обычно включает соглашение об именовании вместе с номерами версий и/или буквенными обозначениями. Подход к идентификации облегчает «хранение, извлечение, отслеживание, воспроизведение и распространение единиц конфигурации» [2].

10.2.2 Базовые версии

В контексте ПО базовая версия – это само ПО и сопровождающие его данные жизненного цикла в определенный момент времени. Базовая версия служит основой для дальнейшей разработки. После установления базовой версии изменения следует вносить только через процесс управления изменениями [2,7].

Базовые версии следует устанавливать на ранних этапах проекта. Однако если слишком рано поместить все элементы под управление конфигурацией, можно навязать лишние процедуры и замедлить работу разработчиков [2]. «До того как программная конфигурация становится базовой версией, изменения можно вносить быстро и неформально. Однако после установления базовой версии изменения по-прежнему возможны, но для оценки и верификации каждого изменения должна применяться конкретная формальная процедура» [4]. Отсюда возникает вопрос: «Когда следует устанавливать базовую версию?» Жесткого универсального правила здесь нет. Обычно базовые версии устанавливаются после каждой фазы жизненного цикла по завершении формального рассмотрения, которым эта фаза заканчивается. Поэтому могут существовать базовая версия требований, базовая версия проектных данных, базовая версия кода и так далее. План управления конфигурацией ПО должен определять подход к установлению базовых версий. Также важно согласовать базовую версию кода с требованиями и проектными данными, которые этот код реализует.

Базовые версии должны быть неизменяемыми. Это означает, что элемент или элементы данных должны быть *зафиксированы* так, чтобы их нельзя было незаметно изменить. Такая неизменяемость важна, потому что необходимо иметь постоянную запись о версии кода и поддерживающих его данных, использованной для формирования выпуска для программных испытаний, летных испытаний, производства и так далее [8].

«Количество и тип базовых версий зависят от того, какую модель жизненного цикла реализует проект. Такие модели жизненного цикла, как спиральная модель, инкрементальная разработка и быстрое прототипирование, требуют большей гибкости при установлении базовых версий» [4].

10.2.3 Трассируемость

Трассируемость тесно связана с базовыми версиями и сообщениями о проблемах. После того как единица конфигурации включена в базовую версию, изменения, как правило, документируются через сообщение о проблеме и/или запрос на изменение. Поэтому при установлении новой базовой версии должна обеспечиваться ее трассируемость к той базовой версии, из которой она произошла.

10.2.4 Сообщения о проблемах

Сообщения о проблемах – одно из важнейших мероприятий управления конфигурацией ПО. Сообщения о проблемах используются для выявления проблем с данными, находящимися в базовой версии, несоответствий процессу и аномального поведения программного обеспечения. Сообщения о проблемах могут создаваться как для устранения проблем, так и для добавления или расширения функциональности ПО. Эффективная работа с сообщениями о проблемах крайне важна для управления изменениями и своевременного исправления известных дефектов. Как отмечалось ранее, работа с сообщениями о проблемах обычно начинается после того, как элемент данных был включен в базовую версию.

См. раздел 9.8, где приведены сводка по содержанию сообщений о проблемах и рекомендации по их составлению и обработке. Сообщения о проблемах ведутся на протяжении всей разработки и верификации программного обеспечения. Проблемы следует расследовать и устранять незамедлительно после их выявления. Кроме того, необходимо прилагать все усилия к тому, чтобы сообщения о проблемах были понятны широкой аудитории, включая разработчиков, руководителей, специалистов по качеству, инженеров по безопасности, системных специалистов, представителей по взаимодействию с сертификацией и сертифицирующие органы.

Ниже приведены несколько тем, связанных с управлением сообщениями о проблемах и особенно важных для сертификации.

10.2.4.1 Управление сообщениями о проблемах при наличии нескольких заинтересованных сторон

В большинстве систем, где активно используется программное обеспечение, участвует несколько заинтересованных сторон. Например, в типичной авионической системе такими сторонами могут быть: изготовитель воздушного судна, системный интегратор, системные инженеры авионики, группа по безопасности, разработчики авионического ПО, поставщик операционной системы, группа верификации ПО и специалисты по сертификации.

У каждой стороны свои приоритеты и своя область компетенции. Когда заинтересованных сторон несколько, может возникнуть ряд проблем, например:

- группы разработки и верификации программного обеспечения могут не понимать системных последствий, последствий для безопасности или для воздушного судна, связанных с частью их проблем;
- у изготовителя воздушного судна или группы системной интеграции может не быть доступа к данным о программных проблемах. Даже если такой доступ есть, у них может не быть достаточной программной подготовки, чтобы полностью понять эти проблемы и принять надлежащие решения по безопасности с учетом рисков для их воздушного

судна и экипажа летных испытаний. Большое количество проблем может скрывать другие проблемы как на уровне программного обеспечения, так и на уровне системы.

Для решения этих и похожих проблем, возникающих при наличии нескольких заинтересованных сторон, предлагаются следующие рекомендации:

Рекомендация 1: согласовать и утвердить подход к классификации сообщений о проблемах. Обычно конечный заказчик, например изготовитель воздушного судна, определяет свою схему классификации проблем для всех поставщиков. Сообщения о проблемах классифицируются по потенциальному влиянию на безопасность, функциональность, производительность, эксплуатацию или обеспечение разработки [10]. Например, у изготовителя воздушного судна могут быть следующие категории серьезности:

1. Проблемы безопасности и соответствия требованиям (должны быть исправлены немедленно)
2. Существенные проблемы функциональности, производительности или эксплуатации (должны быть исправлены до сертификации)
3. Несущественные проблемы функциональности, производительности или эксплуатации (исправить, если возможно)
4. Нефункциональные проблемы, например только документационные (можно отложить)

В своем сертификационном меморандуме CM-SWCEH-002 Европейское агентство по авиационной безопасности (European Union Aviation Safety Agency, EASA) предлагает схему классификации из четырех типов, кратко представленную в таблице 10.1. Термины ошибка (error), отказ (failure) и дефект (fault) определяются следующим образом [11]:

- Ошибка: применительно к программному обеспечению это ошибка в требованиях, проектировании или коде. Дефект: проявление ошибки в ПО. Дефект, если он проявится, может вызвать отказ.
- Отказ: неспособность системы или компонента системы выполнять требуемую функцию в установленных пределах. Отказ может возникнуть при проявлении дефекта. Но дефект также может оставаться скрытым на системном уровне и не иметь эксплуатационных последствий.

Таблица 10.1 Пример схемы классификации сообщений о проблемах

Тип	Потенциальное воздействие	Описание
0	Влияние на безопасность	Проблема, последствием которой при определенных состояниях системы является отказ с неблагоприятным воздействием на безопасность.
1A	Существенный функциональный отказ	Проблема, последствием которой является отказ без неблагоприятного влияния на безопасность системы или оборудования, но с существенным функциональным последствием. Значение слова «существенный» должно определяться в контексте конкретной системы.
1B	Несущественный функциональный отказ	Проблема, последствием которой является отказ без неблагоприятного влияния на безопасность системы или оборудования и без существенного функционального последствия. Значение слов «существенный» и «несущественный» должно определяться в контексте конкретной системы.
2	Нефункциональная неисправность	Проблема, представляющая собой неисправность, которая не приводит к отказу.
3A	Существенное отклонение от планов или стандартов	Любая проблема, которая не относится к типам 0, 1 или 2, но является отклонением от планов или стандартов ПО. Тип 3A – существенное отклонение, последствия которого могут снижать уверенность в том, что ПО ведет себя как задумано и не содержит непредусмотренного поведения.
3B	Несущественное отклонение от планов или стандартов	Любая проблема, которая не относится к типам 0, 1 или 2, но является отклонением от планов или стандартов ПО. Тип 3B – несущественное отклонение, не влияющее на полученную гарантию.
4	Все прочие типы проблем	Все прочие сообщения о проблемах, которые не попадают в предыдущие классификации типов 0-3. Из-за взаимного исключения этих классификаций проблемы типа 4 не имеют функционального последствия. Во многих случаях такие проблемы можно описать как типографические ошибки.

Источник: European Aviation Safety Agency, *Software aspects of certification*, Certification Memorandum CM-SWCEH-002, Issue 1, August 2011.

Кроме того, у программной команды может быть и дополнительная схема приоритетов, помогающая управлять сообщениями о проблемах. Она может включать, например, следующие категории:

- должно быть исправлено немедленно;
- должно быть исправлено до полета (проблема безопасности);
- должно быть исправлено до сертификации (функциональная проблема);
- документационная проблема или проблема не в коде (исправить, если возможно; допустимо отложить);
- отменено.

Схемы классификации всех участников необходимо согласовать, утвердить и описать в планах, включая план управления конфигурацией ПО и планы системного уровня.

Рекомендация 2: согласовывать реализацию, отмену и закрытие сообщений о проблемах. Действия по сообщениям о проблемах, их отмена и закрытие должны быть согласованы с заинтересованными сторонами.

Рекомендация 3: согласовывать передачу сообщений о проблемах вниз по иерархии. Любые сообщения о проблемах, связанные с программным обеспечением и обнаруженные на более высоком уровне, например на уровне воздушного судна или системы, должны передаваться программной команде для обработки. Обычно при этом открывается отдельное сообщение о проблеме по ПО.

Рекомендация 4: согласовывать передачу сообщений о проблемах вверх по иерархии. Все разработчики программного обеспечения должны своевременно предоставлять свои сообщения о проблемах заказчику и организовывать обратную связь.

Необходимо будет добиться согласия относительно критичности проблемы и плана ее устранения. Координация может осуществляться несколькими способами. Сообщения о проблемах могут предоставляться заказчиком, база данных сообщений о проблемах может быть открыта для заказчиков и(или) может действовать совет по рассмотрению сообщений о проблемах с участием заказчика. Для сложных систем с несколькими заинтересованными сторонами очень эффективными могут быть еженедельные или проводимые раз в две недели совещания по рассмотрению сообщений о проблемах. В авиационном проекте в число заинтересованных сторон могут входить представители авиационных систем, авионических систем, летных испытаний, программного обеспечения, безопасности, обес-

печения качества ПО и сертификации. Такая широкая группа заинтересованных сторон необходима потому, что то, что с одной точки зрения кажется несущественной проблемой, с другой точки зрения вполне может оказаться проблемой безопасности. Участие специалистов из разных дисциплин помогает обеспечить тщательный анализ и понимание всех проблем, чтобы с ними можно было правильно работать. Сообщения о проблемах следует предоставлять всем заинтересованным сторонам на рассмотрение до совещания по их рассмотрению; это также дает тем, у кого есть конфликт по расписанию, возможность передать свои замечания или направить представителя.

Сертифицирующие органы выпускали документы с постановкой вопросов и документы по политике, посвященные управлению сообщениями о проблемах при наличии нескольких заинтересованных сторон, например приказ FAA 8110.49 [10] и меморандум EASA CM-SWCEH-002 [11]. Некоторые из этих рекомендаций все еще уточняются. Обязательно учитывайте все конкретные вопросы, которые сертифицирующий орган отметил применительно именно к вашей программе.

10.2.4.2 Управление открытыми и отложенными сообщениями о проблемах

Крайне желательно и настоятельно рекомендуется, чтобы сообщения о проблемах были обработаны и закрыты до сертификации. Обоснование открытых сообщений о проблемах и управление ими во время сопровождения может стать для проекта серьезной дополнительной нагрузкой и источником проблем в последующих проектах модернизации.

Недавно я консультировала проект, в котором было 2000 проваленных тестов из-за неоднозначного требования. Разработчик хотел отложить сообщение о проблеме. Однако обосновать 2000 непройденных тестов – задача не из легких. После долгих споров все согласились, что проще и аккуратнее исправить требование и добиться прохождения тестов, чем пытаться обосновать, почему допустимо сертифицировать систему с 2000 непройденных тестов.

Большинство проектов в итоге имеют хотя бы несколько отложенных сообщений о проблемах. Если это так, в отложенных сообщениях о проблемах должны быть представлены тщательный анализ и обоснование того, почему отсрочка не влияет на безопасность или соответствие требованиям. Проблемы, влияющие на безопасность или соответствие требованиям, нельзя откладывать. Решение об отсрочке должно быть понятно и согласовано всеми заинтересованными сторонами, а также кратко изложено в итоговом отчете разработки ПО (Software Accomplishment Summary, SAS) для согласования с сертифицирующим органом. В главе 12 объясняется, что такое итоговый отчет разработки ПО и что именно в нем ожидается в отношении любых отложенных сообщений о проблемах.

Большое количество отложенных сообщений о проблемах создает трудности по нескольким причинам:

- это может указывать на незрелость системы и вызывать сомнения в достаточности обеспечения разработки. Кроме того, трудно оценить взаимодействие проблем между собой;
- трудно должным образом обосновать большое количество проблем с точки зрения безопасности и соответствия требованиям;
- может быть трудно утверждать, что программное обеспечение выполняет свои предусмотренные функции, как того требуют нормативные документы;
- крайне трудно сопровождать такие сообщения о проблемах после выпуска. Каждый раз, когда программное обеспечение меняется или используется в другой установке, сообщения о проблемах необходимо оценивать заново.

Все это следует учитывать, принимая решение, исправлять проблему или откладывать ее.[*]

10.2.5 Управление изменениями и их рассмотрение

Управление изменениями и их рассмотрение тесно связаны с процессом сообщений о проблемах, поскольку такие сообщения часто используются как средство управления изменениями. Изменения могут возникать на любой стадии процесса разработки ПО и фактически являются его неизбежным свойством. Их вносят, чтобы добавить или изменить функции, исправить ошибку, повысить производительность, обновить аппаратуру и связанные с ней интерфейсы, улучшить процессы, изменить среду, например перейти на более новый компилятор, и так далее. Изменять ПО относительно легко, и это одно из главных преимуществ его использования. Но управлять этими изменениями совсем не так просто. Без эффективного управления изменениями начинается хаос. Управление изменениями сложно, но необходимо. Эффективный процесс управления изменениями включает несколько задач, в том числе следующие:

1. *Защита от несанкционированных изменений.* После установления базовой версии она должна быть защищена от непреднамеренных и несанкционированных изменений. Все изменения базовой версии должны быть запланированы, документированы и утверждены, как правило, через совет по управлению изменениями.
2. *Инициирование изменения.* Все предлагаемые изменения базовой версии должны быть документированы. Обычным средством документирования предлагаемого изменения является сообщение о проблеме или запрос на изменение.
3. *Классификация изменения.* Как отмечалось ранее в разделе 10.2.4, изменения классифицируются по их влиянию на общую безопасность системы, функциональность и соответствие требованиям.
4. *Оценка влияния изменения.* Чтобы правильно спланировать ресурсы, необходимые для

реализации изменения, и объем последующей повторной верификации, влияние изменения документируют в сообщении о проблеме или запросе на изменение. Следует отметить, что если программное обеспечение уже прошло формальные испытания или сертификацию, обычно требуется более формальный анализ влияния изменения, как обсуждается далее в этой главе.

5. *Рассмотрение изменения.* В разделе 7.1.е стандарта DO-178С указано, что цель рассмотрения изменений состоит в том, чтобы обеспечить, что «проблемы и изменения оцениваются, одобряются или отклоняются, одобренные изменения реализуются, а затронутые процессы получают обратную связь посредством методов сообщения о проблемах и управления изменениями, определенных в процессе планирования программного обеспечения» [5]. Рассмотрение изменений включает рассмотрение всех изменений, чтобы определить их возможное влияние. Обычно роль «привратника» в процессе изменений выполняет совет по управлению изменениями. Этот совет рассматривает все сообщения о проблемах и запросы на изменение, одобряет или отклоняет предлагаемые изменения, подтверждает, что одобренные изменения надлежащим образом реализованы и верифицированы, а также подтверждает, что сообщения о проблемах или запросы на изменение должным образом обновляются в ходе разработки и верификации и закрываются после завершения работ. Поэтому в совет по управлению изменениями должны входить участники, хорошо знающие ПО и систему, а также обладающие полномочиями и знаниями для принятия решений по изменениям. Нередко заказчик, например изготовитель воздушного судна и/или группа интеграции авионики, входит в состав такого совета. Ранее упоминался совет по рассмотрению сообщений о проблемах. В большинстве проектов совет по рассмотрению сообщений о проблемах и совет по управлению изменениями – это один и тот же орган. Однако в некоторых крупных проектах они могут быть разделены.
6. *Решение по изменению.* Совет по управлению изменениями оценивает предлагаемое изменение и одобряет его, отклоняет, откладывает или возвращает на доработку, если нужна дополнительная информация.
7. *Реализация изменения.* После того как предлагаемое изменение одобрено советом по управлению изменениями, изменение вносится так, как было согласовано в сообщении о проблеме или запросе на изменение. Если в ходе реализации объем изменения меняется, сообщение о проблеме или запрос на изменение необходимо обновить; кроме того, может потребоваться дополнительное рассмотрение советом по управлению изменениями. Как отмечалось ранее, изменение единицы конфигурации должно приводить к изменению ее идентификации, обычно обновляется версия или редакция, но иногда требуется изменить обозначение. Кроме того, как отмечалось ранее, изменения про-

граммного обеспечения должны быть трассируемы от их источника. Энн Хасс пишет: «Для любой единицы конфигурации должна существовать возможность идентифицировать изменения в ней относительно ее предшественника. Любое изменение должно быть трассируемо до элемента, в котором это изменение было реализовано» [6].

8. *Верификация изменения.* После реализации изменения проводится его верификация. Обычно для этого проводят независимое рассмотрение всех измененных или затронутых артефактов, а также повторно выполняют все необходимые тесты.
9. *Закрытие запроса на изменение или сообщения о проблеме.* После того как изменение реализовано, верифицировано и принято, его можно закрыть. Как правило, за закрытие сообщения о проблеме или запроса на изменение отвечает совет по управлению изменениями.
10. *Выпуск обновленной базовой версии.* Базовая версия не должна выпускаться до тех пор, пока все одобренные сообщения о проблемах или запросы на изменение не будут верифицированы.

10.2.6 Учет состояния конфигурации

Учет состояния конфигурации включает регистрацию и представление информации, необходимой для эффективного управления разработкой и верификацией ПО. Отчеты формируются для информирования руководителей, разработчиков и других заинтересованных сторон о состоянии проекта. Учет состояния конфигурации дает согласованную, надежную, своевременную и актуальную информацию о состоянии проекта. Она помогает улучшить коммуникацию, избежать дублирования и не повторять одни и те же ошибки [2]. Он часто включает отчеты, содержащие следующее:

- Состояние элементов данных, включая идентификацию конфигурации.
- Состояние сообщений о проблемах и запросов на изменение, включая классификацию, затронутые элементы данных, корневую причину проблем и идентификацию конфигурации обновленных элементов данных.
- Состояние выпущенных данных и файлов.
- Перечень содержимого базовой версии и отличий от предыдущей базовой версии.

Для автоматизации отчетов по учету состояния конфигурации часто используются инструменты управления конфигурацией. Чтобы такие инструменты были успешны, данные должны вводиться точно и последовательно. Кроме того, во избежание ошибочного использования пользователи должны хорошо понимать функциональность инструмента.

Отчеты о состоянии конфигурации следует планировать на ранних этапах проекта, чтобы

данные собирались надлежащим образом. Однако по мере развития проекта иногда оказывается желательно изменить или расширить метрики. Например, в сводку по сообщениям о проблемах можно добавить поля для идентификации персонала (кто выявил проблему, какая команда вызвала проблему, кто исправил проблему и так далее), чтобы лучше понимать потребности в персонале.

Отчеты о состоянии следует обновлять с определенной периодичностью. Большинство компаний либо автоматизируют их, чтобы они всегда были актуальны, либо обновляют еженедельно перед совещаниями проектной команды. Устаревший или ошибочный отчет бесполезен, а иногда и ведет к неверным решениям. Недавно я участвовала в проекте, который регулярно предоставлял устаревшую и неточную сводку по сообщениям о проблемах. Мне, как представителю Федерального управления гражданской авиации США, было очень трудно понять, на какие данные опираться при оценке влияния на безопасность. Ошибочные данные также заставили меня усомниться в точности других данных. Поэтому крайне важно прилагать все усилия к формированию точных отчетов о состоянии; иначе они могут принести больше вреда, чем пользы.

10.2.7 Выпуск

Когда элемент данных достигает достаточной зрелости, он выпускается в формальную систему управления конфигурацией (обычно в библиотеку масштаба всей организации). DO-178C определяет *release* (выпуск) как: «Действие, формально открывающее доступ к воспроизводимой единице конфигурации и разрешающее ее использование» [5].

Не все данные должны проходить через формальный процесс выпуска. Некоторые данные можно просто хранить и защищать. Обычно формальному выпуску подлежат данные, необходимые для повторной сборки и сопровождения программного обеспечения (такие как требования, проект, код, файлы конфигурации и указатель конфигурации). Поддерживающие данные, такие как записи рассмотрений и записи по гарантии качества ПО, достаточно хранить вместе с проектной документацией, как правило на защищенном сервере. DO-178C определяет минимальный набор элементов данных, подлежащих выпуску, с использованием категоризации КК1/КК2. Это будет подробнее рассмотрено позже, когда мы перейдем к категориям контроля в этой главе (раздел 10.2.9).

Процесс выпуска обычно включает рассмотрение и утверждение документа или данных ключевыми заинтересованными сторонами (например, автором, рецензентом, инженером по качеству программного обеспечения, представителем по взаимодействию с сертификацией и специалистом по качеству данных). Прежде чем подписывать или утверждать элемент данных, его действительно нужно прочитать. Вероятно, это кажется очевидным, но мне встречался не один подписанный документ, который никто не читал.

Обычно данные (включая исполняемый код) выпускаются до передачи заказчику и до использования для получения сертификационного зачета.

10.2.8 Архивирование и извлечение

Процесс архивирования и извлечения включает хранение данных (как выпущенных данных, так и других данных, используемых для поддержки сертификации) таким образом, чтобы к ним могли получить доступ уполномоченные лица. В процессе архивирования и извлечения следует учитывать следующее [5]:

- *Точность и полноту.* Необходимо удостовериться, что архивируются именно те данные, которые требуется архивировать. Часто это делают при подготовке и рассмотрении указателя конфигурации ПО (Software Configuration Index, SCI), чтобы убедиться, что данные в SCI совпадают с тем, что было заархивировано.
- *Защиту от несанкционированного изменения.* Данные должны обновляться только уполномоченными лицами и только при наличии надлежащего разрешения на изменение.
- *Качество носителя хранения.* Его нужно учитывать, чтобы минимизировать ошибки восстановления в краткосрочной перспективе и деградацию данных в долгосрочной. Не всякий носитель подходит для хранения критичного по безопасности программного обеспечения.
- *Защиту при авариях и бедствиях.* Обычно для этого используют тот или иной вариант хранения вне основной площадки.
- *Читаемость данных.* Данные типовой конструкции должны оставаться доступными и читаемыми все время, пока оборудование установлено на воздушном судне. В зависимости от надежности носителя это может требовать периодического обновления данных.
- *Архивирование поддерживающих инструментов.* Если для чтения или генерации данных требуются инструменты, их тоже может потребоваться архивировать; это может включать среду разработки и среду верификации. Если инструменты требуют лицензионных соглашений, это тоже следует учитывать.
- *Корректность извлечения и копирования.* При извлечении или копировании данные не должны повреждаться.
- *Способность обрабатывать изменения без потери данных.* При изменении данных ранее выпущенные и заархивированные данные не должны затрагиваться.

10.2.9 Категории управления данными

Чтобы определить, какой объем управления конфигурацией требуется для каждого типа данных, в DO-178C используется понятие *control category* (категория контроля). Концепция категорий контроля специфична для авиации и объясняется в DO-178C. Определены две категории контроля: категория контроля 1 (КК1) и категория контроля 2 (КК2). Таблицы приложения А стандарта DO-178C указывают применимость КК1 или КК2 для каждого элемента данных.[*]

КК1 требует максимального уровня управления и применяется к ключевым сертификационным данным и данным, необходимым для повторной генерации или расследования происшествий. Кроме того, для уровней А и В большее число элементов данных классифицируется как КК1. Для КК1 требуется применение следующих процессов управления конфигурацией: идентификация конфигурации, базовые версии, трассируемость, сообщения о проблемах, управление изменениями (целостность, идентификация и отслеживание), рассмотрение изменений, учет состояния конфигурации, извлечение, защита от несанкционированных изменений, выбор носителей, обновление данных, копирование, выпуск и хранение данных [5]. КК2 является подмножеством КК1 и требует ограниченного управления конфигурацией. Она применяется к элементам данных, не столь критичным для повторной генерации исполняемого объектного кода, например к записям управления конфигурацией и записям верификации. Для КК2 требуется применение следующих процессов управления конфигурацией: идентификация конфигурации, трассируемость к источнику, управление изменениями (целостность и идентификация), извлечение, защита от несанкционированных изменений и хранение данных [5].

Таблица 10.2 обобщает требования DO-178C по КК1 и КК2 для различных элементов данных. Некоторые элементы данных всегда относятся к КК1: План сертификации ПО (PSAC), требования, данные трассируемости разработки, исходный код, исполняемый объектный код, файлы элементов параметрических данных, Указатель конфигурации ПО (SCI) и Итоговый отчет разработки ПО (SAS). Аналогично, некоторые элементы данных всегда относятся к КК2: сообщения о проблемах, записи по гарантии качества ПО, результаты верификации и записи управления конфигурацией ПО. Для остальных элементов данных категория контроля зависит от уровня ПО.

Таблица 10.2 Категории управления по типам данных

Данные ЖЦ ПО по DO-178C (русский аналог: КТ-178С)	Раздел DO- 178C	A	B	C	D
План сертификации ПО (PSAC)	11.1	KK1	KK1	KK1	KK1
План разработки ПО (SDP)	11.2	KK1	KK1	KK2	KK2
План верификации ПО (SVP)	11.3	KK1	KK1	KK2	KK2
План управления конфигурацией ПО (SCMP)	11.4	KK1	KK1	KK2	KK2
План гарантии качества ПО (SQAP)	11.5	KK1	KK1	KK2	KK2
Стандарты требований к ПО	11.6	KK1	KK1	KK2	N/A
Стандарты проектирования ПО	11.7	KK1	KK1	KK2	N/A
Стандарты кодирования ПО	11.8	KK1	KK1	KK2	N/A
Данные требований высокого уровня к ПО	11.9	KK1	KK1	KK1	KK1
Описание проекта ПО (SDD)	11.10	KK1	KK1	KK1	KK2a
Исходный код	11.11	KK1	KK1	KK1	KK1
Исполняемый объектный код	11.12	KK1	KK1	KK1	KK1
Тестовые примеры и процедуры верификации ПО (SVCP)	11.13	KK1	KK1	KK2	KK2a
Результаты верификации ПО (SVR)	11.14	KK2	KK2a	KK2a	KK2a
Указатель конфигурации среды ЖЦ ПО (SECI)	11.15	KK1	KK1	KK1	KK2
Указатель конфигурации ПО (SCI)	11.16	KK1	KK1	KK1	KK1
Сообщения о проблемах (PR)	11.17	KK2	KK2	KK2	KK2
Записи управления конфигурацией ПО	11.18	KK2	KK2	KK2	KK2
Записи гарантии качества ПО	11.19	KK2	KK2	KK2	KK2a
Итоговый отчёт разработки ПО (SAS)	11.20	KK1	KK1	KK1	KK1
Данные трассировки разработки	11.21	KK1	KK1	KK1	KK1
Данные трассировки верификации	11.21	KK1	KK1	KK2	KK2a
Файл компонента параметрических данных	11.22	KK1	KK1	KK1	KK1

а Только когда требуются данные жизненного цикла ПО, что зависит от конкретной цели. Назначения категорий контроля в DO-178C считаются минимально необходимыми. Многие компании предпочитают выйти за эти пределы и относят к КК1 больше элементов, чем требует DO-178C. Например, отчет по верификации ПО нередко рассматривают как КК1, хотя DO-178C относит результаты верификации ПО к КК2.

10.2.10 Управление загрузкой

Исполняемый код ничего не делает, пока он фактически не загружен в целевую аппаратуру. Часть ПО загружается на заводе, а часть – в эксплуатации. (См. главу 18, где рассматривается ПО, загружаемое в эксплуатации.) Ниже приведены ключевые составляющие управляемого процесса загрузки ПО:

- *Утвержденные процедуры загрузки.* Хотя сама загрузка часто находится вне рамок разработки ПО и области действия DO-178C, поскольку относится к производству или техническому обслуживанию воздушного судна, ее необходимо учитывать в процессе одобрения ПО. Процедуры загрузки должны быть разработаны и верифицированы в рамках работ по обеспечению соответствия DO-178C. Раздел 11.16.k стандарта DO-178C рекомендует включать процедуры загрузки в указатель конфигурации ПО, так же как и инструкции по сборке. Это было добавлено в DO-178C по сравнению с DO-178B. В DO-178B не было ясно, где именно должны документироваться процедуры загрузки.
- *Верификация загрузки.* Должен существовать способ убедиться, что ПО загружено полностью и без повреждений. Часто это делают с помощью проверки целостности, например контрольной суммы CRC (cyclic redundancy check, циклический избыточный код).
- *Верификация маркировки ПО.* Должен существовать способ идентифицировать загруженное ПО, чтобы подтвердить, что его обозначение и версия соответствуют одобренным данным. После загрузки ПО следует проверить, что его маркировка согласуется с одобренными данными.
- *Совместимость с аппаратурой.* Одобренная совместимость аппаратуры и ПО должна быть документирована, и ей необходимо следовать.

10.2.11 Управление средой жизненного цикла ПО

Среда жизненного цикла ПО включает методы, инструменты, процедуры, языки программирования и аппаратуру, используемые для разработки, верификации, управления и формирования данных жизненного цикла ПО и самого программного продукта [5]. Среда, которая не находится под управлением, может привести к множеству проблем: ошибкам в коде, потере элементов данных, трудноуловимым отказам при тестировании, недостаточному тести-

рованию, невозпроизводимому процессу сборки и так далее. «Управление средой жизненного цикла ПО гарантирует, что инструменты, используемые для создания программного обеспечения, идентифицированы, находятся под управлением и доступны для извлечения» [5].

На этапе планирования среда разработки ПО описывается в Плане разработки ПО (SDP), среда верификации описывается в Плане верификации ПО (SVP), среда управления конфигурацией описывается в Плане управления конфигурацией ПО (SCMP), а среда гарантии качества ПО описывается в Плане гарантии качества ПО (SQAP). В этих планах описываются процедуры, инструменты, методы, стандарты и аппаратура, используемые для реализации процессов. Также рекомендуется сводно перечислить все инструменты в Плане сертификации ПО (PSAC).

Однако на этапе планирования детали среды часто еще неизвестны (например, может быть известен компилятор, но конкретная версия, параметры или настройки и поддерживающие библиотеки на этапе планирования могут быть еще неизвестны). Чтобы среда была детерминированной, детали, включая конкретные исполняемые файлы, должны быть документированы. Подробности среды жизненного цикла ПО фиксируются в Указателе конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index, SECI) или в эквивалентном документе. Типовое содержимое Указателя конфигурации среды ЖЦ ПО будет описано позже (см. раздел 10.4.3). Этот указатель используется для управления средой. Должны существовать процессы, подтверждающие, что инженеры действительно используют среду, указанную в Указателе конфигурации среды ЖЦ ПО. Об этой задаче часто забывают. Указатель конфигурации среды ЖЦ ПО нередко завершают только к концу программы. Однако, чтобы среда действительно находилась под управлением, он должен быть подготовлен на раннем этапе процесса. Поскольку среды разработки, управления конфигурацией и гарантии качества ПО могут быть известны раньше, чем среда верификации, Указатель конфигурации среды ЖЦ ПО может выпускаться в нескольких итерациях.

Кроме того, инструменты, используемые для разработки, верификации, управления, сборки и загрузки программного обеспечения и его данных, должны находиться под конфигурационным управлением. Следовательно, инструменты должны управляться как данные категории КК1 или КК2. DO-178C требует, чтобы исполняемый объектный код неквалифицированных инструментов управлялся как минимум по категории КК2. Соответствующая категория контроля для квалифицированных инструментов и поддерживающих их квалификационных данных определяется в DO-330, *Software Tool Qualification Considerations*. Квалификация инструментов рассматривается в главе 13; однако о среде использования квалифицированных инструментов стоит предупредить заранее. Если в вашем проекте используется квалифицированный инструмент, важно проверить, что среда, в которой вы его

используете (эксплуатационная среда), репрезентативна относительно среды, в которой инструмент был квалифицирован. Иначе зачет по квалификации инструмента может оказаться неприменимым.

10.3 Специальные навыки в управлении конфигурацией ПО

Для управления конфигурацией важен опытный и квалифицированный персонал. Управление конфигурацией – обязанность всех участников, однако руководитель управления конфигурацией или библиотекарь управления конфигурацией и члены совета по управлению изменениями играют в этом процессе особую роль.

Библиотекарь управления конфигурацией ПО создает библиотеку или библиотеки управления конфигурацией ПО и обеспечивает целостность каждой библиотеки, а также целостность между библиотеками [6]. Эта роль может поддерживаться средствами автоматизации. Библиотекарь управления конфигурацией ПО должен обладать следующими навыками: пониманием общих потребностей компании в управлении конфигурацией, вниманием к деталям, способностью документировать подробные процедуры, способностью обеспечивать соблюдение процедур, умением хорошо взаимодействовать с разными людьми и пониманием используемых средств автоматизации.

Совет по управлению изменениями отвечает за оценку изменений, одобрение или отклонение изменений и контроль выполнения всех согласованных действий. Члены этого совета должны понимать продукт, возможное влияние изменений и общую систему управления конфигурацией ПО. Руководитель или председатель совета должен уметь координировать людей с разными характерами, проводить совещания, доводить согласованные действия до выполнения и взаимодействовать на разных уровнях. Сюда относится и способность учитывать потребности различных заинтересованных сторон при определении влияния изменений, приоритетов и вопросов, связанных с безопасностью.

10.4 Данные управления конфигурацией ПО

Таблица А-8 стандарта DO-178С определяет цели процесса управления конфигурацией ПО и данные, формируемые в ходе этого процесса. В этом разделе кратко поясняется каждый элемент данных жизненного цикла управления конфигурацией ПО.

10.4.1 План управления конфигурацией ПО (SCMP)

Содержание Плана управления конфигурацией ПО описано в главе 5. План управления конфигурацией ПО следует разрабатывать на раннем этапе проекта, чтобы управление конфигурацией применялось надлежащим образом. План должен охватывать вопросы из этой главы, а также вопросы, описанные в главе 5. Помимо самого плана, для некоторых мероприятий управления конфигурацией ПО могут потребоваться более подробные процедуры.

Например, процесс регистрации сообщений о проблемах и процесс анализа воздействия изменений нередко требуют конкретных деталей сверх того, что содержится в плане управления конфигурацией ПО.

10.4.2 Сообщения о проблемах

Сообщения о проблемах уже рассматривались ранее в этой главе и в главе 9. Их нужно идентифицировать и поддерживать в актуальном состоянии на протяжении всего проекта. Также важно иметь качественную сводку состояния сообщений о проблемах: она нужна для управленческих решений по проекту, а также для сертификационных оценок и оценок безопасности.

10.4.3 Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index, SECI)

Как отмечалось ранее, а также в разделе 11.15 стандарта DO-178C, указатель конфигурации среды ЖЦ ПО используется, чтобы документировать среду разработки, сборки, загрузки, верификации и управления и держать ее под контролем. Он поддерживает повторную генерацию, повторную верификацию и модификацию программного обеспечения. В него включается перечень следующих элементов [5]:

- Аппаратуры и операционной системы среды разработки.
- Инструментов, используемых для разработки, сборки и загрузки программного обеспечения (таких как компиляторы, компоновщики, загрузчики, инструменты управления требованиями).
- Инструментов, используемых для верификации программного обеспечения, включая аппаратуру для испытаний и операционные системы.
- Инструментов, используемых процессами управления конфигурацией ПО и гарантии качества ПО.
- Квалифицированных инструментов и данных, используемых для поддержки квалификации.

Во многих проектах Указатель конфигурации среды ЖЦ ПО объединяют с Указателем конфигурации ПО в один пакет. При таком подходе нужно учитывать, что потребуются несколько итераций Указателя конфигурации ПО: среда должна находиться под управлением еще до сборки и выпуска кода.

10.4.4 Указатель конфигурации ПО (Software Configuration Index, SCI)

Указатель конфигурации ПО содержит перечень всех данных жизненного цикла программного продукта в конкретной конфигурации, включая исходный код и файлы исполняемого объектного кода. Если программное обеспечение содержит несколько компонентов, может

существовать иерархия таких указателей; например, отдельный указатель для каждого компонента и затем указатель верхнего уровня, объединяющий все компоненты.

Указатель конфигурации ПО является важным элементом данных, потому что он, по сути, «идентифицирует конфигурацию программного продукта» [5] и важен для сертификации и сопровождения. Этот указатель определяет, что именно представляет собой программное обеспечение и какие данные использовались для его разработки. Он также является одним из трех элементов данных, которые требуется представлять сертифицирующему органу, наряду с Планом сертификации ПО (PSAC) и Итоговым отчетом разработки ПО (SAS).

Согласно разделу 11.16 стандарта DO-178C, указатель конфигурации ПО идентифицирует программный продукт: обозначение или обозначения ПО, загружаемого в оборудование, исполняемый объектный код, файлы исходного кода с информацией о версиях, архивные и выпускные носители, данные жизненного цикла, инструкции по сборке для получения исполняемого объектного кода, ссылку на указатель конфигурации среды ЖЦ ПО или его включение в состав документа, используемые проверки целостности данных, например контрольную сумму CRC, процедуры и методы загрузки, а также процедуры и методы для модифицируемого пользователем ПО, если оно используется [5]. Если применяются файлы компонентов параметрических данных, они также перечисляются вместе с инструкциями по сборке для их формирования. Файлы компонентов параметрических данных рассматриваются в главе 22.

Обычно указатель конфигурации ПО формируется для каждой базовой версии программного обеспечения. Для базовых версий разработки такой указатель часто содержит лишь перечень файлов исходного кода, инструкции по сборке, закрытые сообщения о проблемах и запросы на изменение, а также ссылку на данные указателя конфигурации среды ЖЦ ПО или их включение. Однако для каждой базовой версии также рекомендуется указывать версии планов, стандартов, требований и проекта, которые использовались для формирования кода. Остальные данные могут добавляться по мере развития проекта, но эти элементы важны для понимания того, что именно определяло код.

Обычно я просматриваю несколько Указателей конфигурации ПО в год. Есть две тенденции, которые я часто вижу; ниже они приведены, чтобы вы могли заранее их учитывать:

- Часто Указатель конфигурации ПО включает листинг кода, но не включает остальные данные жизненного цикла ПО. Это особенно часто встречается в компаниях с военным прошлым: они привыкли разрабатывать Version Description Document (документ описания версии), который требовался стандартом DOD-STD-2167A и другими военными стандартами.
- Многие команды предпочитают объединять указатель конфигурации среды ЖЦ ПО и

указатель конфигурации ПО в одном документе. Это допустимо и даже прямо упоминается в DO-178C. Однако команды часто не завершают документ до момента выпуска программного обеспечения. Если Указатель конфигурации среды ЖЦ ПО входит в состав Указателя конфигурации ПО, потребуется ранняя версия документа для определения среды разработки. Иначе трудно доказать, что среда действительно находилась под управлением.

10.4.5 Записи управления конфигурацией ПО

Записи управления конфигурацией ПО включают дополнительные данные, используемые в этом процессе, такие как отчеты учета состояния, записи о выпусках, записи об изменениях, записи библиотеки ПО и записи о базовых версиях. Конкретный набор формируемых записей различается от компании к компании. План управления конфигурацией ПО должен либо идентифицировать данные, которые будут формироваться, либо ссылаться на процедуры компании, где эти детали определены.

10.5 Подводные камни управления конфигурацией ПО

В управлении конфигурацией ПО есть несколько подводных камней, которых следует избегать. Одни глубже других, но проблемы могут создать все. Некоторые вопросы уже упоминались ранее, однако здесь они приведены снова, чтобы дать сводный перечень.

Подводный камень 1: не охвачены все требуемые мероприятия. Необходимы все мероприятия управления конфигурацией ПО, упомянутые в разделе 10.2. Если какие-то из них отсутствуют, это быстро может привести к хаосу.

Подводный камень 2: неправильное планирование. Хорошее управление конфигурацией ПО не возникает само собой. Оно требует планирования и детализированных процедур. Планы и процедуры должны охватывать как инженерное, или разработческое, управление конфигурацией, так и общеорганизационное, или формальное, управление конфигурацией.

Подводный камень 3: отсутствие понимания, приверженности или поддержки со стороны руководства. Управление конфигурацией ПО требует ресурсов, включая инструменты, обучение и персонал. Без поддержки руководства ресурсы обычно ограничены, и потому процесс управления конфигурацией ПО не дотягивает до того уровня, который действительно нужен.

Подводный камень 4: отсутствие квалифицированного персонала. Управление конфигурацией ПО требует специализированного набора навыков, как отмечалось в разделе 10.3. Отсутствие людей с такими навыками приводит к неэффективному управлению конфигурацией.

Подводный камень 5: неправильное использование автоматизации. Один только инстру-

мент не решит проблемы управления конфигурацией в организации. Как выразилась Джессика Киз, «автоматизация процесса, приносящего убытки, позволяет терять больше денег быстрее и с большей точностью» [9]. Существует множество инструментов управления конфигурацией, от бесплатных до очень дорогих. Одни рассчитаны на большие программные команды, другие лучше подходят для небольшой команды. Одни хороши для формального управления конфигурацией и долговременного хранения, другие лучше работают для повседневных инженерных нужд. Для некоторых проектов может быть полезно использовать и долговременный, и повседневный набор инструментов. Следует очень внимательно подходить к тому, что именно автоматизировать, и к выбору инструмента для такой автоматизации.

Подводный камень 6: недостаток обучения. Чтобы управление конфигурацией применялось последовательно, все участники должны понимать процедуры. Обучение по управлению конфигурацией должно быть обязательным для всех членов команды. Оно не обязано занимать много времени. Компьютерное обучение может быть вполне эффективным.

Подводный камень 7: среда не находится под контролем. Как уже отмечалось, неконтролируемые среды разработки и верификации – распространенная проблема программных проектов. Указатель конфигурации среды ЖЦ ПО должен разрабатываться своевременно и поддерживаться в актуальном состоянии, чтобы все инженеры использовали корректную среду. Кроме того, настройки всех инструментов должны быть где-то задокументированы, обычно в Указателе конфигурации среды ЖЦ ПО или в разделе Указателя конфигурации ПО с инструкциями по сборке, чтобы гарантировать правильную настройку и использование инструментов.

Подводный камень 8: процесс учета сообщений о проблемах недостаточно определен или внедрен. Часто планы и процедуры не объясняют, когда начинается процесс учета сообщений о проблемах и как члены команды должны его выполнять. В результате возникают ситуации, когда проблемы не документируются должным образом, не исправляются своевременно, исправляются без документации, не оцениваются на предмет воздействия, не координируются между заинтересованными сторонами и так далее.

Подводный камень 9: сообщения о проблемах написаны не для более широкой аудитории. Очень часто сообщения о проблемах пишутся так, что их не может нормально читать широкий круг людей, которым нужно их понимать, например руководители, заказчик и сертифицирующий орган.

Подводный камень 10: изменения вносятся без разрешения. Иногда, пока инженер делает свою работу, он замечает другую проблему и решает ее исправить. Само по себе это не плохо, если только никто больше не поставлен в известность об этом изменении и оно не

задокументировано. После того как данные включены в базовую версию, все изменения должны документироваться в сообщении о проблеме или в запросе на изменение либо в эквивалентном процессе.

Подводный камень 11: непонимание и отсутствие контроля управления конфигурацией у поставщика. У одних поставщиков могут быть хорошо выстроенные процессы управления конфигурацией, у других этот процесс может находиться в зачаточном состоянии. Важно понимать процессы управления конфигурацией всех поставщиков, включая субподрядчиков и офшорные команды, и как можно раньше разбираться с любыми ограничениями, слабостями или несовместимостями. Ниже приведены примеры того, что следует учитывать при работе с поставщиками:[*]

- Как поставщик идентифицирует свои данные и совместимо ли это со схемой идентификации конфигурации вашей компании? Важно понимать, как именно он именуется или нумерует свои элементы данных. Для данных, поставляемых поставщиком, может потребоваться добавить номер вашей компании либо хранить данные в отдельной библиотеке или со специальным атрибутом.
- Какие данные будут поставляться и какие данные будут сопровождаться самим поставщиком? После поставки данных кто отвечает за их сопровождение? Кто отвечает за хранение каких данных и где именно?
- Что произойдет, если поставщика продадут другой компании или он прекратит работу? Как организовано управление изменениями данных поставщика? Будет ли у него собственный совет по управлению изменениями? Если да, то какой уровень прозрачности будет предоставлен в отношении работы этого совета? Какой учет состояния предоставляет поставщик? Какой процесс учета сообщений о проблемах будет использовать поставщик? Как поставщик управляет своей средой?

Подводный камень 12: не подтверждена эксплуатационная среда квалифицированных инструментов. Когда квалифицированные инструменты используются для поддержки разработки или верификации критичного по безопасности программного обеспечения, важно подтвердить, что инструмент используется в той эксплуатационной среде, для которой он был квалифицирован. Каждый инструмент квалифицируется для одной или нескольких эксплуатационных сред. Пользователи инструмента должны подтвердить, что используют его именно в такой среде. Иначе инструмент может работать некорректно. В некоторых случаях неправильная работа может быть неочевидной. Квалификация инструментов рассматривается в главе 13.

Подводный камень 13: среда не архивируется. Среду, то есть аппаратные средства, операционную систему и поддерживающие программные инструменты, может потребоваться ар-

хивировать наряду с требованиями, проектом, исходным кодом и исполняемым объектным кодом, чтобы поддерживать повторную генерацию, расследование происшествий, сохранение летной годности и так далее.

Подводный камень 14: процедуры не находятся под контролем. Во многих компаниях существуют общеорганизационные процедуры, размещенные во внутренней сети для удобного доступа всей организации. Иногда планы ссылаются на такие процедуры, но конфигурационное управление ими ограничено. У процедур может не быть статуса редакции; они могут обновляться без уведомления и не архивироваться, так что предыдущие версии оказываются недоступны для извлечения. Команда программного обеспечения должна работать по известному набору процедур. Для этого может потребоваться зафиксировать комплект процедур в отдельной рабочей области или использовать дополнительный процесс управления конфигурацией сверх общеорганизационной внутренней сети.

Подводный камень 15: не учитывается долговременная читаемость данных. Для целей сопровождения следует учитывать носитель и формат данных. Помимо сохранения данных на надежном носителе, таком как компакт-диск (compact disk, CD) или цифровые видеодиски (digital video disk, DVD), должно быть доступно оборудование и программное обеспечение для чтения этого носителя. По этой причине рекомендуется хранить данные в бумажной форме или в формате .pdf, чтобы их можно было прочитать в будущем. Это особенно важно, когда для просмотра данных требуются специализированные инструменты, особенно если такой инструмент требует ежегодного продления контракта или лицензии.

Подводный камень 16: процедуры сборки и/или загрузки не являются повторяемыми. Как отмечалось ранее, процедуры сборки и загрузки программного обеспечения документируются в Указателе конфигурации ПО. Нередко единственный человек, который способен выполнить сборку или загрузку, это тот, кто занимается этим ежедневно. Поскольку этот человек не будет доступен круглосуточно на протяжении всего срока эксплуатации оборудования, процедуры должны быть документированы так, чтобы их можно было повторить. Их должен выполнить кто-то, не знакомый с этими процедурами, чтобы убедиться, что процедуры понятны и что при их выполнении получаются те же результаты, то есть они повторяемы.

10.6 Анализ воздействия изменений

Изменения программного обеспечения – обычная часть жизни критичных по безопасности систем. Несколько лет назад на одной конференции производитель систем управления двигателем заявил, что более 90% их работ по разработке ПО состоит из изменений уже летающего ПО.

В авиационной отрасли производные продукты составляют подавляющую часть нашей ра-

боты. В таких производных продуктах обычно повторно используют максимально возможную часть ПО, добавляя новые функции или исправления, когда это необходимо. Поэтому важно проектировать ПО так, чтобы его было удобно сопровождать и модернизировать.

Когда в критичных по безопасности системах происходят изменения программного обеспечения, нужна особая осторожность. Изменения в уже эксплуатируемом ПО должны быть тщательно спланированы, проанализированы, реализованы, поставлены под контроль и верифицированы или испытаны. Кроме того, необходимо принять меры, чтобы убедиться, что изменение не окажет отрицательного влияния на другие функции в системе или на воздушном судне.

Процесс изменения рассматривался в разделе 10.2.5. Каждое изменение в ходе разработки документируется в сообщении о проблеме или в запросе на изменение. Для каждого такого сообщения или запроса на изменение рассматривается влияние этого изменения. Затем каждый измененный или затронутый программный компонент и соответствующие данные жизненного цикла верифицируются до включения в сборку ПО и до закрытия сообщения о проблеме или запроса на изменение. После ввода ПО в эксплуатацию анализ воздействия изменений (Change Impact Analysis, CIA) становится более формальным процессом, требующим дополнительной документации. Следует отметить, что такая формальность может потребоваться и в ходе первоначальной разработки, если формальные испытания уже завершены, а изменение требует регрессионного тестирования. В 2000 году FAA впервые опубликовало политику по анализу воздействия изменений. С тех пор соответствующие указания оставались стабильными. Такой анализ требуется для ПО, уже одобренного в составе сертифицированного изделия, если его модифицируют, например для исправления ошибки, добавления функциональности или адаптации под другого пользователя. При его выполнении следует учитывать несколько целей:

- Он помогает команде разработки и верификации программного обеспечения определить объем повторной работы.
- Он помогает совету по управлению изменениями определить, какие изменения одобрять, а какие нет.
- Он дает способ оценить потенциальное влияние изменения на безопасность.
- Он дает заказчику информацию для поддержки его анализа воздействия изменений на уровне системы или воздушного судна.
- Он служит средством обоснования отнесения изменения к незначительным или значительным.

Обычно предварительный анализ воздействия изменений выполняется на ранней стадии ра-

бот по модификации программного обеспечения, чтобы оценить объем работ, необходимых для реализации изменения, определить потенциальное влияние изменения на безопасность и согласовать с сертифицирующим органом классификацию изменения как значительного или незначительного. Предварительный анализ воздействия изменений обычно либо документируется как часть Плана сертификации ПО для обновленного ПО, либо включается в отдельный документ, на который ссылаются документы планирования ПО. В конце проекта этот анализ обновляется, чтобы отразить фактически выполненный анализ, и часто включается в Итоговый отчет разработки ПО (SAS).

Согласно FAA Order 8110.49 изменения программного обеспечения, не имеющие потенциального влияния на безопасность, могут классифицироваться как незначительные, тогда как изменения, которые могут повлиять на безопасность, обычно классифицируются как значительные [10]. И значительные, и незначительные изменения проходят один и тот же процесс, однако значительные изменения требуют участия сертифицирующего органа.

В главе 11 FAA Order 8110.49[*] перечислены элементы, которые как минимум должны оцениваться в составе анализа воздействия изменений [10]. Каждый элемент нужно рассмотреть, даже если в итоге он не окажет никакого влияния; если он неприменим, в соответствующем разделе анализа можно написать «Неприменимо» и объяснить почему. Ниже кратко перечислены виды анализа, которые включаются в анализ воздействия изменений [10,12]:

- *Анализ трассируемости* выявляет требования, проектные элементы, код, а также тестовые примеры и процедуры, которые могут быть прямо или косвенно затронуты изменением. Прямая трассируемость изменений выявляет компоненты проектирования, затронутые изменением. Обратная трассируемость помогает определить другие конструктивные особенности и требования, которые могут быть непреднамеренно затронуты изменением. В целом трассируемость требований помогает определить влияние изменения на проект ПО. Важно выявить как измененные, так и затронутые программное обеспечение и данные.
- *Анализ запаса памяти* удостоверяет, что требования к распределению памяти по-прежнему выполняются, исходная карта памяти сохранена и поддерживаются достаточные запасы памяти. Этот анализ обычно невозможно завершить, пока изменение не реализовано. На раннем этапе его можно оценить приблизительно, но фактическая оценка воздействия выполняется позже.
- *Анализ временных запасов* подтверждает, что временные требования, включая требования к расписанию и интерфейсам, по-прежнему выполняются, характеристики конкуренции за ресурсы понятны и временные запасы по-прежнему поддерживаются. Как и

анализ запаса памяти, анализ времени обычно невозможно завершить, пока изменение не реализовано.

- *Анализ потоков данных* выявляет любые неблагоприятные эффекты из-за изменений потоков данных, а также связанность между компонентами и внутри них. Для выполнения анализа потоков данных каждая переменная и каждый интерфейс, затронутые изменением, должны быть проанализированы, чтобы убедиться, что исходная инициализация этой переменной остается корректной, что изменение внесено последовательно и что оно не влияет на любое другое использование этого элемента данных. Если используются большие глобальные области данных, такой процесс может оказаться дорогим и сложным.
- *Анализ потока управления* выявляет любые неблагоприятные эффекты из-за изменений потока управления и связанности компонентов. Примерами учитываемых элементов являются планирование задач, порядок выполнения, приоритизация и структура прерываний.
- *Анализ ввода-вывода* удостоверяет, что изменения не оказали отрицательного влияния на требования к входам и выходам продукта. Рассматриваются такие аспекты, как загрузка шин, доступ к памяти, пропускная способность, аппаратные входы и интерфейсы устройств вывода.
- *Анализ среды разработки и процессов* выявляет любые изменения, которые могут повлиять на программный продукт, например изменение параметров или версии компилятора и изменение оптимизации, изменение инструкций или параметров компоновщика, ассемблера и загрузчика, либо изменение программного инструмента. Следует также учитывать целевое аппаратное обеспечение. Если меняется процессор или иное аппаратное обеспечение, взаимодействующее с программным обеспечением, это может повлиять на способность ПО выполнять предназначенную функцию.
- *Анализ эксплуатационных характеристик* рассматривает изменения, которые могут повлиять на работу системы, например изменение отображений и символов, параметров производительности, коэффициентов усиления, фильтров, ограничений, проверок данных, обработки прерываний и исключений, а также смягчения отказов, чтобы убедиться в отсутствии неблагоприятных эффектов.
- *Анализ требований к поддержанию летной годности* определяет, вызывают ли изменения программного обеспечения необходимость в новых или измененных требованиях к поддержанию летной годности. В ходе первоначальной сертификации авиационного изделия определяются требования к поддержанию летной годности. Например, тормоза может потребоваться осматривать после 100 посадок. Если изменение ПО влияет

на требование к поддержанию летной годности, это должно быть учтено в процессе изменения.

- *Анализ разделения* удостоверяет, что изменения не влияют на защитные механизмы, включенные в проект. Если в составе схемы разделения применяются архитектурные меры смягчения, изменение не должно затрагивать эти стратегии. Например, данные не должны передаваться из менее критичного раздела в более критичный раздел, если только более критичный раздел не выполняет надлежащую проверку этих данных.

Анализ воздействия изменений также документирует данные жизненного цикла ПО, такие как требования, проект, архитектура, исходный и объектный код, тестовые примеры и процедуры, которые были изменены или затронуты изменением, а также действия по верификации, например рассмотрения, анализы, инспекции и тесты, необходимые для того, чтобы гарантировать отсутствие неблагоприятных воздействий на систему в результате изменения.

Организованный, документированный и тщательный процесс анализа воздействия изменений важен для планирования и реализации проекта, повторной верификации и регрессионного тестирования (дополнительную информацию о регрессионном тестировании см. в главе 9), а также для сертификации. Компании, разрабатывающие критичное по безопасности программное обеспечение, должны иметь общеорганизационный процесс анализа воздействия изменений, который удовлетворяет потребности проекта, поддерживает безопасность и соответствует потребностям сертифицирующего органа. Чтобы это работало, анализ должен выявлять все измененные и затронутые данные, включая само ПО, верификацию измененных и затронутых данных, а также анализ влияния на безопасность. Этот анализ должен быть читаемым и всеобъемлющим; он должен учитывать политику и руководящие указания сертифицирующего органа, например FAA Order 8110.49 или EASA Certification Memorandum CM-SWCEH-002. Предварительный анализ может быть более высокоуровневым, но итоговый анализ должен ясно указывать, какие данные были изменены или затронуты, какая верификация была проведена, каковы итоговые характеристики ПО по сравнению с исходными характеристиками, например временные характеристики и запасы памяти, и каким образом были выявлены и верифицированы любые воздействия на безопасность. Следует отметить, что предварительный анализ воздействия изменений не только поддерживает усилия по сертификации, но и служит хорошим инструментом управления проектом для планирования ресурсов, бюджета и графика обновления ПО.

Общеорганизационный процесс должен также определять подход к документированию предварительных и итоговых анализов воздействия изменений. То есть нужно объяснить, входят ли они в состав Плана сертификации ПО (PSAC) и Итогового отчёта разработки ПО

(SAS), оформляются как самостоятельные документы или используются иные варианты. Процесс должен также определять, кто утверждает такие анализы и передаются ли они в сертифицирующий орган. Например, некоторые организации требуют, чтобы уполномоченный FAA представитель рассматривал и утверждал такой анализ. Если изменение признано значительным, анализ включается в состав Плана сертификации ПО (PSAC) и Итогового отчёта разработки ПО (SAS) и передается в FAA; однако если изменение незначительное, анализ может существовать отдельно от Плана сертификации ПО и Итогового отчёта разработки ПО и в FAA не направляться.

Многие компании выполняют анализ воздействия изменений для каждого сообщения о проблеме или запроса на изменение. Это хорошая практика; однако такой анализ должен учитывать и совокупное воздействие сообщений о проблемах и/или запросов на изменение.

Ссылки

1. W.A. Babich, *Software Configuration Management* (Chichester, U.K.: John Wiley & Sons, 1991).
2. A. Leon, *Software Configuration Management Handbook*, 2nd edn. (Norwood, MA: Artech House, 2005).
3. A. Lager, The evolution of configuration management standards, *Logistics Spectrum*, Huntsville, AL, January-March 2002.
4. R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 7th edn. (New York: McGraw-Hill, 2010).
5. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
6. A.M.J. Haas, *Configuration Management Principles and Practice* (Boston, MA: Addison-Wesley, 2003).
7. IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std610-1990 (Los Alamitos, CA: IEEE Computer Society Press, 1990).
8. B. Aiello and L. Sachs, *Configuration Management Best Practices* (Boston, MA: Addison-Wesley, 2011).
9. J. Keyes, *Software Configuration Management* (Boca Raton, FL: CRC Press, 2004).
10. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Change 1, September 2011).
11. European Aviation Safety Agency, *Software Aspects of Certification*, Certification Memorandum CM-SWCEH-002 (Issue 1, August 2011).

12. L. Rierson, A systematic process for changing safety-critical software, IEEE Digital Avionics Systems Conference (Philadelphia, PA: 2000).

*Глава 14 FAA Order 8110.49 [10] и глава 16 EASA CM-SWCEH-002 [11] содержат рекомендации по управлению открытыми сообщениями о проблемах.

*Таблицы приложения А документов DO-330, DO-331, DO-332 и DO-333 также указывают КК1 или КК2 для каждого элемента данных.

*Это не исчерпывающий перечень, а лишь отправная точка.

*После публикации DO-178С и его дополнений политика FAA, вероятно, будет обновлена. Сам процесс анализа воздействия изменений, вероятно, останется очень похожим, однако если от вас требуется выполнять такой анализ, следует подтвердить, что вы используете самую свежую политику.

Глава 11. Гарантия качества ПО

Сокращения

Сокращение	Расшифровка
CMMI	Интеграция модели зрелости возможностей
IEC	Международная электротехническая комиссия
IEEE	Институт инженеров по электротехнике и электронике
ISO	Международная организация по стандартизации
PDCA	Планируй, делай, проверяй, действуй
PQA	Гарантия качества продукта
PQE	Инженер по качеству продукта
PR	Сообщение о проблеме
PSAC	План сертификации ПО
SAS	Итоговый отчет разработки ПО
SCI	Указатель конфигурации ПО
SCM	Управление конфигурацией ПО
SCMP	План управления конфигурацией ПО
SEI	Институт программной инженерии
SECI	Указатель конфигурации среды ЖЦ ПО
SQA	Гарантия качества ПО
SQAP	План гарантии качества ПО
SQE	Инженер по гарантии качества ПО
SVCP	Тестовые примеры и процедуры верификации ПО
SVR	Результаты верификации ПО

11.1 Введение: качество ПО и гарантия качества ПО (Software Quality Assurance, SQA)

11.1.1 Определение качества программного обеспечения

Чтобы отвечать потребностям области ПО, критически важного для безопасности, разработчики должны последовательно заботиться о качестве. Высококачественные продукты не возникают сами по себе. Они появляются в хорошо управляемых организациях, где качество действительно важно, и благодаря работе талантливых, добросовестных и дисциплинированных инженеров.

American Heritage Dictionary включает в определение качества следующее: «неотъемле-

мая или отличительная характеристика», «существенная характеристика» и «степень или уровень совершенства» [1]. В программной инженерии существует несколько взглядов на то, что такое качество, но обычно преобладают два. Первая – точка зрения разработчика: продукт качественный, если программное обеспечение соответствует установленным для него требованиям. Вторая – точка зрения заказчика: продукт качественный, если ПО удовлетворяет его потребности. Продукт, который соответствует установленным требованиям, но не удовлетворяет потребности заказчика, заказчик не сочтет высококачественным. Требования критически важны для устранения разрыва между взглядом разработчика и взглядом заказчика на качество. Требования должны разрабатываться так, чтобы удовлетворять потребности заказчика, чтобы разработчики могли создать и верифицировать ПО, соответствующее этим требованиям. Как отмечалось в главе 6, важно привлекать заказчика к процессу определения требований. Без тесного взаимодействия заказчика и разработчика на этапе определения требований качество остается труднодостижимой целью.

Роджер Прессман пишет: «Можно сделать это правильно, а можно потом переделывать. Если команда по разработке программного обеспечения уделяет внимание качеству во всех мероприятиях программной инженерии, она уменьшает объем переделок, которые ей придется выполнять. Это приводит к снижению затрат и, что еще важнее, к сокращению времени вывода продукта на рынок» [2]. «Программа качества представляет собой структуру, позволяющую встроить качество в продукт, выполнять оценки, необходимые для определения того, работает ли эта структура, и оценивать качество, фактически достигнутое в продукте» [3]. На качество влияют мероприятия на протяжении всего жизненного цикла, включая определение требований, проектирование, кодирование, верификацию и сопровождение.

11.1.2 Характеристики высококачественного программного обеспечения

Для определения целей высококачественного программного обеспечения часто используют атрибуты качества. К таким атрибутам относятся корректность, эффективность, гибкость, функциональность, целостность, интероперабельность, сопровождаемость, переносимость, повторное использование, тестируемость и удобство использования. Стандарт Международной организации по стандартизации (ISO) и Международной электротехнической комиссии (IEC) 9126 (ISO/IEC 9126) определяет набор атрибутов качества, называемых характеристиками и подхарактеристиками. Эти характеристики и подхарактеристики кратко приведены ниже [4,5]:

- **Функциональность:** способность программного продукта предоставлять функции, которые удовлетворяют заявленные и подразумеваемые потребности при использовании программного обеспечения в заданных условиях. Подхарактеристики функциональности включают пригодность, точность, интероперабельность, защищенность и соответ-

ствие функциональности. *Надежность*: способность программного продукта поддерживать заданный уровень работоспособности при использовании в заданных условиях. Подхарактеристики надежности включают зрелость, устойчивость к неисправностям, восстанавливаемость и соответствие надежности.

- *Удобство использования*: способность программного продукта быть понятным, изучаемым, используемым и привлекательным для пользователя при использовании в заданных условиях. Подхарактеристики удобства использования включают понятность, обучаемость, управляемость, привлекательность и соответствие удобству использования.
- *Эффективность*: способность программного продукта обеспечивать надлежащую производительность относительно объема используемых ресурсов при заданных условиях. Подхарактеристики эффективности включают поведение во времени, использование ресурсов и соответствие эффективности.
- *Сопровождаемость*: способность программного продукта быть изменяемым. Изменения могут включать исправления, улучшения или адаптацию программного обеспечения к изменениям среды, требований и функциональных спецификаций. Подхарактеристики сопровождаемости включают анализируемость, изменяемость, стабильность, тестируемость и соответствие сопровождаемости. *Переносимость*: способность программного продукта переноситься из одной среды в другую. Подхарактеристики переносимости включают адаптируемость, устанавливаемость, сосуществование, заменяемость и соответствие переносимости.

Компании часто внедряют метрики для измерения подмножества этих атрибутов, хотя многие из них на практике довольно трудно выразить количественно.

11.1.3 Гарантия качества ПО

Большинство компаний внедряют процесс гарантии качества ПО, чтобы убедиться, что требуемые атрибуты качества достигнуты, а программное обеспечение соответствует своим требованиям. «Формальное определение гарантии качества ПО состоит в том, что это систематические мероприятия, предоставляющие свидетельства пригодности всего программного продукта к использованию» [6]. Прессман пишет: «Гарантия качества создает инфраструктуру, поддерживающую надежные методы программной инженерии, рациональное управление проектом и действия по контролю качества... Кроме того, гарантия качества состоит из набора функций аудита и отчетности, которые оценивают результативность и полноту действий по контролю качества» [2]. Мероприятия по гарантии качества ПО дают уверенность, что установлены и соблюдаются адекватные процессы, позволяющие выпускать продукты, соответствующие требованиям и потребностям заказчика.

Процесс гарантии качества ПО по DO-178C (КТ-178C) является интегральным процессом, который непрерывно выполняется на протяжении планирования ПО, разработки, верификации и финальных работ по подтверждению соответствия. Один или несколько инженеров по гарантии качества ПО обеспечивают разработку планов и стандартов и их соблюдение в ходе реализации. Инженер по гарантии качества ПО также выполняет рассмотрение соответствия ПО, чтобы подтвердить полноту данных и соблюдение требований.

Хотя DO-178C требует наличия процесса гарантии качества ПО, следует отметить, что качество не является исключительно ответственностью персонала гарантии качества ПО. Кроме того, качество программного обеспечения нельзя просто оценить в конце проекта, как это делается в некоторых инженерных дисциплинах. Как отмечает Уильям Льюис: «Качества нельзя добиться путем оценки уже заверченного продукта. Поэтому цель состоит в том, чтобы прежде всего предотвращать дефекты или недостатки качества и делать продукты пригодными для оценки средствами гарантии качества... В дополнение к оценкам продукта, для программы гарантии качества существенны и оценки процесса» [6]. Эмануэль Бейкер и Мэтью Фишер выражают ту же мысль: «Хотя оценочные мероприятия необходимы, сами по себе они не обеспечат требуемое качество. То есть качество продукта нельзя привнести в него посредством оценки, тестирования, аудита, анализа, измерений или инспекции. Качество можно только встроить в ходе процесса разработки» [3]. Качество – это непрерывная работа и ответственность всей программной команды. DO-178C поощряет качество через использование стандартов и мероприятий по верификации на протяжении всего ЖЦ ПО. Качество должно быть встроено в продукт; его нельзя добавить аудитом, надзором или тестированием.

DO-178C требует, чтобы все цели гарантии качества ПО выполнялись с соблюдением независимости. Для гарантии качества ПО DO-178C дает немного иное определение независимости, чем для верификации. DO-178C определяет независимость так:

Разделение ответственности, гарантирующее выполнение объективной оценки. (1) Для мероприятий процесса верификации ПО независимость достигается тогда, когда верификационное мероприятие выполняется лицом или лицами, не участвовавшими в разработке верифицируемой единицы, а также инструментом или инструментами, обеспечивающими эквивалентные человеку верификационные мероприятия. (2) Для процесса гарантии качества ПО независимость также включает полномочия для обеспечения выполнения корректирующих действий [7].

Первая половина этого определения относится к независимости верификации, которая обсуждалась в главе 9. Вторая половина относится к процессу гарантии качества ПО и к его месту в организационной структуре. Хотя это и не требуется, большинство компаний вы-

полняют требование независимости за счет отдельной службы гарантии качества ПО, которая не находится в подчинении у инженерного подразделения. В этом разделе под гарантией качества ПО понимается группа, ответственная за выполнение соответствующих мероприятий. Такая группа может быть отдельной организацией, что предпочтительно, включающей одного или нескольких инженеров по гарантии качества ПО, либо отдельной функцией в составе инженерной организации, что обычно встречается только в очень небольших компаниях.

DO-178C поощряет процессно-ориентированный подход к гарантии качества ПО. Гарантия качества ПО в первую очередь отвечает за то, чтобы процессы, определенные в утвержденных планах и стандартах, соблюдались. Однако некоторые организации начинают понимать, что одного процесса недостаточно. Некоторые компании принимают концепцию гарантии качества продукта (Product Quality Assurance, PQA) Института программной инженерии (Software Engineering Institute, SEI), при которой инженер отвечает за качество продукта, а не только процесса. При таком подходе инженер по качеству продукта (Product Quality Engineer, PQE) и инженер по гарантии качества ПО тесно взаимодействуют, чтобы обеспечивать и качество продукта, и качество процесса. Оценки процесса демонстрируют, что процессы соблюдались, тогда как оценки продукта демонстрируют корректность выходных данных процесса. Следует отметить, что именно такой всегда была концепция верификации в DO-178B, а теперь и в DO-178C; однако такие инженеры помогают преодолеть разрыв между инженерной организацией и службой гарантии качества ПО. Использование этой роли требует, чтобы специалист обладал знаниями предметной области и опытом по разрабатываемому продукту.

ISO разработала ряд документов, определяющих надлежащие практики гарантии качества ПО, включая семейство документов, называемое *ISO 9000* [*] и включающее ISO 9000, 9001 и 9004. ISO 9000 задает терминологию и основы серии стандартов по системам качества. ISO 9001 объединяет более ранние стандарты, известные как ISO 9001, 9002 и 9003. ISO 9001 является общим и может применяться к любому продукту. ISO 9001 включает следующие базовые элементы [2]:

- Установить элементы системы управления качеством. Документировать систему качества.
- Поддерживать контроль качества и обеспечение качества.
- Установить механизмы рассмотрения системы управления качеством.
- Определить ресурсы качества, включая персонал, обучение и элементы инфраструктуры.

- Определить методы устранения проблем.

Большинство разработчиков программного обеспечения для систем, критичных к безопасности, проходят регистрацию по ISO 9000 и поддерживают ее. Регистрацию по ISO 9000 выполняет аккредитованный сторонний орган. Надзор проводится каждые 6 месяцев. Повторная регистрация требуется каждые 3 года [5]. Другие общепромышленные организации и стандарты, такие как модель интеграции зрелости возможностей (Capability Maturity Model Integration, CMMI)(R) Института программной инженерии и стратегия Six Sigma, впервые популяризированная Motorola в 1980-х годах, также предоставляют основу для гарантии качества ПО.

11.1.4 Примеры распространенных проблем качества процесса и продукта

В работах по разработке программного обеспечения может возникать множество проблем качества. В таблице 11.1 кратко приведены некоторые распространенные проблемы процесса и продукта. Очевидно, что большинство проблем процесса, если их заранее не устранять, могут приводить к проблемам продукта. Поэтому важно учитывать как качество процесса, так и качество продукта.

Таблица 11.1 Распространенные проблемы процесса и продукта

Распространенные проблемы процесса	Распространенные проблемы продукта
• Рассмотрения требований и проекта пропускаются либо выполняются людьми без необходимых технических навыков или понимания системы. • Процесс сборки не является повторяемым; каждый раз при сборке программного обеспечения получается другой результат. • В исходный код добавляются изменения для исправления некорректной функциональности, но требования и проектные данные не обновляются, чтобы отразить изменение. • Конфигурация данных ЖЦ ПО, включая исходный код, не поддерживается. • Планы и стандарты не соблюдаются. • Среды разработки и верификации не определены.	• Заказчику поставляется дефектное ПО. • ПО не удовлетворяет всем требованиям. • ПО недостаточно устойчиво; в обычной ситуации оно работает нормально, но отказывает, когда происходит что-то нештатное. • Требования неполны или неоднозначны; поэтому продукт не соответствует ожиданиям или потребностям заказчика. • Исходный код не соответствует требованиям и проектным данным.

11.2 Характеристики эффективной и неэффективной гарантии качества ПО

11.2.1 Эффективная гарантия качества ПО

Назначение гарантии качества ПО уже было объяснено, а соответствующие мероприятия будут описаны чуть позже. Прежде чем разбирать, чем занимается служба гарантии качества ПО, стоит описать некоторые предпосылки эффективной гарантии качества ПО. «Качество никогда не бывает случайностью; оно всегда является результатом разумных усилий» [8]. Для эффективного внедрения гарантии качества ПО необходимы следующие характеристики.

Характеристика 1: поддержка со стороны высшего руководства. Чтобы гарантия качества ПО была эффективной, она должна иметь одобрение, приверженность и поддержку со

стороны высшего руководства. Это должна быть искренняя приверженность, а не просто декларации.

Высшее руководство должно быть привержено общему качеству своих продуктов и предоставлению ресурсов, необходимых для укомплектования и обучения службы гарантии качества ПО. Эту концепцию настойчиво подчеркивали такие специалисты по качеству, как Каору Исикава, Джозеф М. Джуран, У. Эдвардс Деминг и Уоттс С. Хамфри.

Характеристика 2: независимость. Эффективная гарантия качества ПО независима от организаций разработки и верификации. Независимость помогает уменьшить ошибки, возникающие из-за чрезмерной вовлеченности в повседневный процесс или продукт, который оценивается. Кроме того, независимость помогает обеспечивать выполнение корректирующих действий, когда это необходимо. Без независимости возникает давление, иногда намеренное, а иногда нет, приводящее к компромиссам при жестких ограничениях по бюджету и графику.

Характеристика 3: техническая компетентность. Чтобы служба гарантии качества ПО была эффективной и заслуживала уважения команд разработки и верификации, она должна быть укомплектована хорошо квалифицированными и опытными техническими инженерами. Без технической силы служба гарантии качества ПО не будет своевременно выявлять реальные проблемы и не будет пользоваться уважением инженерной организации.

Характеристика 4: обучение. Персонал гарантии качества ПО, как и вся инженерная организация в целом, должен быть надлежащим образом обучен ожиданиям в области качества программного обеспечения. Обучение должно быть непрерывным, чтобы ошибочные установки или модели поведения устранялись или хотя бы сводились к минимуму, а новые сотрудники должным образом вводились в курс дела. Персонал гарантии качества ПО должен проходить обучение в следующих областях: разработка и верификация ПО, навыки аудита или оценки и DO-178C. Преподавая по всему миру курсы по DO-178B, а затем по DO-178C более 15 лет, я видела не более десяти сотрудников гарантии качества ПО среди слушателей. Увы, большинство компаний не вкладываются в обучение своих инженеров по гарантии качества ПО, и общее качество их ПО отражает этот недостаток инвестиций.

Характеристика 5: непрерывное совершенствование. Организации с хорошим качеством всегда стремятся к улучшению. Деминг рекомендует цикл «планируй, делай, проверяй, действуй» (Plan-Do-Check-Act, PDCA), который обычно называют *циклом Деминга* или *кругом Деминга*, поскольку это циклический процесс. Идея состоит в том, чтобы разработать план, а затем выполнить его. Мероприятие контролируется, то есть проверяется, и затем предпринимаются действия по улучшению процесса. Этот цикл является непрерывным, чтобы постоянно повышать общую результативность процесса и качество создаваемых продуктов.

11.2.2 Неэффективная гарантия качества ПО

За многие годы автора расстраивало отсутствие хороших процессов гарантии качества ПО во многих организациях. При анализе того, почему гарантия качества ПО оказывается неэффективной, почти всегда обнаруживается, что причина связана с провалом в одной из пяти отмеченных выше областей, то есть в поддержке со стороны высшего руководства, независимости, технической компетентности, обучении или непрерывном совершенствовании. Сбой в одной или нескольких из этих областей приводит к следующим проблемам:

- *Недоукомплектованная служба гарантии качества ПО.* Если руководство не поддерживает службу гарантии качества ПО, она часто оказывается слишком малочисленной, чтобы выполнять свою работу.
- *Неквалифицированные инженеры по гарантии качества ПО.* Привлечь хороших инженеров в службу гарантии качества ПО непросто. Многие из сильнейших инженеров хотят проектировать программное обеспечение, а не разбирать чужой проект ПО. Поэтому руководителям службы гарантии качества ПО приходится проявлять изобретательность, чтобы привлекать хорошие кадры. Некоторые компании используют ротацию сроком на 1 или 2 года как ступень к управлению проектами. Некоторые компании также используют модель расширения полномочий, при которой персонал гарантии качества ПО обучает сотрудников и возглавляет корпоративную приверженность качеству. Когда люди понимают некоторые преимущества работы в службе гарантии качества ПО, вероятность их прихода в команду возрастает. В таблице 11.2 кратко перечислены квалификация и характеристики, на которые стоит обращать внимание при найме инженеров по гарантии качества ПО.

Таблица 11.2 Квалификации и характеристики эффективных инженеров по гарантии качества ПО

Подготовка, навыки и личные качества инженера по гарантии качества ПО
От трех до пяти лет опыта в разработке или верификации; образование в области инженерии или компьютерных наук; опыт в различных аспектах разработки, например кодирование, написание требований, тестирование.
Хорошие письменные и устные коммуникативные навыки, поскольку инженеры по гарантии качества ПО взаимодействуют с руководством, инженерами и персоналом по взаимодействию с сертифицирующим органом.
Готовность развиваться в сторону руководящей должности или позиции менеджера программы, поскольку работа в гарантии качества ПО дает хорошее понимание компании в целом.
Способность справляться со сложными ситуациями.
Готовность отстаивать правильное решение даже под давлением.
Надежность, то есть способность внушать доверие и взаимодействовать с несколькими заинтересованными сторонами, не поддаваясь давлению.
Независимость, но готовность принимать указания, то есть способность работать при минимальном надзоре и при этом уважать полномочия руководства.
Увлеченность качеством и безопасностью.
Обучаемость и стремление учиться.

- *Игнорируемая и потому неэффективная служба гарантии качества ПО.* Когда в службе гарантии качества ПО работают неквалифицированные или неэффективные сотруд-

ники, инженерная организация не уважает замечания этой службы и не реагирует на них.

11.3 Мероприятия по гарантии качества ПО

В этом разделе описаны типовые задачи, которые служба гарантии качества ПО выполняет силами одного или нескольких инженеров по качеству ПО, чтобы обеспечить соблюдение планов, стандартов и процессов компании.

Задача 1: рассмотрение планов. Служба гарантии качества ПО рассматривает План сертификации ПО (Plan for Software Aspects of Certification, PSAC), планы по ПО и стандарты разработки, оценивая соответствие целям DO-178C и согласованность между планами [7].[*] В главе 5 приводится информация о планах и стандартах. Большинство компаний проводят коллегиальное рассмотрение планов, и служба гарантии качества ПО участвует в этом рассмотрении. В рамках работ по рассмотрению контрольные перечни обычно заполняются как проектной командой, так и службой гарантии качества ПО. Помимо планов по ПО, служба гарантии качества ПО должна разбираться и в планах уровня системы и безопасности, поскольку они могут влиять на программные процессы.

Задача 2: написание Плана ГК ПО. Команда гарантии качества ПО отвечает за подготовку Плана гарантии качества ПО (Software Quality Assurance Plan, SQAP) и поддержание его в актуальном состоянии. В большинстве компаний существует общий корпоративный план гарантии качества ПО. Однако у каждого проекта могут быть свои уникальные особенности, которые обычно детализируются в проектном плане гарантии качества ПО или в плане сертификации ПО.

В главе 5 описано, что обычно включается в план гарантии качества ПО. Кроме того, для определения его состава можно использовать и другие источники. Например, стандарт IEEE 730 Института инженеров по электротехнике и электронике определяет типовые компоненты, рекомендуемые для включения в план гарантии качества ПО [9]. В таблице 11.3 кратко приведены предлагаемые стандартом IEEE 730 разделы.

Таблица 11.3 Краткое содержание стандарта IEEE 730

Название раздела	Описание раздела
Назначение	Объясняет назначение плана гарантии качества ПО.
Ссылочные документы	Перечисляет все документы, на которые ссылается план.
Управление	Объясняет организационную структуру, задачи и ответственность по проекту.

Название раздела	Описание раздела
Документация	Определяет документацию, используемую для задания разработки, верификации и конфигурации ПО.
Стандарты, практики, соглашения и метрики	Определяет стандарты, практики, соглашения и метрики, которые должны применяться, и описывает, как будет контролироваться и обеспечиваться соблюдение.
Рассмотрения и аудиты	Определяет рассмотрения, выполняемые технической командой, а также план участия службы гарантии качества ПО в рассмотрениях и аудите процессов.
Тестирование	Объясняет подход и методы тестирования, а также роль службы гарантии качества ПО в наблюдении за тестированием и надзоре.
Сообщения о проблемах и корректирующие действия	Описывает процесс регистрации сообщений о проблемах и роль службы гарантии качества ПО.
Инструменты, методы и методологии	Определяет и объясняет любые инструменты, методы или методологии, которые служба гарантии качества ПО использует для выполнения своей роли.
Контроль кода	Объясняет процесс документирования, сопровождения, хранения, выпуска и извлечения программного обеспечения.
Контроль носителей	Определяет методы идентификации носителя для каждого продукта.
Контроль поставщиков	Объясняет роль службы гарантии качества ПО в контроле поставщиков и надзоре за ними.
Сбор записей, сопровождение и хранение	Объясняет подход службы гарантии качества ПО к документированию.
Обучение	Объясняет необходимое обучение для персонала гарантии качества ПО, а также для инженеров проекта.

Название раздела	Описание раздела
Управление рисками	Объясняет, как риски выявляются и управляются на протяжении проекта.

Источник: Software Engineering Standards Committee of the IEEE Computer Society, *IEEE Standard for Software Quality Assurance Plans*, IEEE Std. 730-2002, IEEE, New York, 2002.

Задача 3: утверждение ключевых элементов данных. Служба гарантии качества ПО обычно утверждает ключевые элементы данных, такие как планы, стандарты, требования, проектные данные, документ с тестовыми примерами и процедурами верификации ПО (Software Verification Cases and Procedures, SVCP), отчет о результатах верификации ПО (Software Verification Results, SVR), Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index, SECI), Указатель конфигурации ПО (Software Configuration Index, SCI), данные квалификации инструмента и Итоговый отчет разработки ПО (Software Accomplishment Summary, SAS). Нередко ключевые элементы данных не могут быть выпущены без утверждения службы гарантии качества ПО.

Задача 4: аудит данных ЖЦ ПО. Служба гарантии качества ПО проводит аудит данных ЖЦ ПО, чтобы удостовериться, что процессы соблюдаются, стандарты применяются, контрольные перечни правильно заполняются и цели DO-178C выполняются.[*] Иногда аудиты проводятся в ходе коллегиальных рассмотрений или совместно с рассмотрениями персонала по взаимодействию с сертифицирующими органами. Однако аудиты службы гарантии качества ПО могут проводиться и независимо от других мероприятий. Некоторые организации гарантии качества ПО используют планы и стандарты для разработки комплекта материалов для аудита программного обеспечения; затем служба гарантии качества ПО выполняет аудит процессов и данных с использованием этого комплекта. Такой подход дает подробные свидетельства того, что проект соблюдает утвержденные планы и стандарты.

Задача 5: участие в рассмотрениях. По своему усмотрению,[†] служба гарантии качества ПО участвует в коллегиальных рассмотрениях на протяжении процессов разработки и верификации программного обеспечения. Служба гарантии качества ПО выявляет технические проблемы, а также проверяет соблюдение инженерной командой процесса коллегиального рассмотрения, заполнение контрольных перечней, выполнение критериев входа и выхода и так далее. Участие в коллегиальных рассмотрениях является хорошим способом для службы гарантии качества ПО оставаться в курсе работ и проблем проекта.

Задача 6: оценка критериев перехода. В таблице А-9 DO-178C, цель 4, указано: «Получены гарантии того, что критерии перехода для процессов ЖЦ ПО выполнены» [7]. Эта цель гарантии качества ПО часто достигается за счет участия службы гарантии качества ПО в

коллегиальных рассмотрениях. Однако у службы гарантии качества ПО может существовать и отдельное мероприятие по оценке критериев перехода.

Задача 7: присутствие при тестировании. Служба гарантии качества ПО часто играет ключевую роль в наблюдении за тестированием ПО. Некоторые организации требуют, чтобы служба гарантии качества ПО наблюдала за определенной долей выполняемых тестов. Эти ожидания должны быть определены в плане гарантии качества ПО и ясно доведены до проекта.

Задача 8: аудит среды. Служба гарантии качества ПО обычно проводит аудит сред разработки и верификации, чтобы убедиться, что они согласованы с утвержденной конфигурацией, обычно задокументированной в указателе конфигурации среды ЖЦ ПО. Перед сборкой ПО для выпуска служба гарантии качества ПО может наблюдать за настройкой машины сборки, как объяснено в разделе 8.4.1, чтобы удостовериться в соблюдении утвержденных процедур и соответствии машины сборки утвержденной конфигурации. Перед формальным тестированием, дающим зачет для сертификации, служба гарантии качества ПО обычно проверяет настройку стенда тестирования или наблюдает за ней, чтобы убедиться в соответствии утвержденной конфигурации.

Задача 9: наблюдение за сборкой и загрузкой. Используя документированные процедуры сборки и загрузки, служба гарантии качества ПО может выполнять сборку и загрузку самостоятельно либо наблюдать за выполнением этих процедур другим лицом, чтобы подтвердить повторяемость. Как объяснялось в главе 8, лучше, чтобы процедуры выполнял человек, который обычно этим не занимается, например инженер по качеству ПО или другой инженер.

Задача 10: участие в совете по контролю изменений. Служба гарантии качества ПО часто играет важную роль в совете по контролю изменений, поскольку она оценивает предлагаемые изменения и анализирует реализацию этих изменений. Для закрытия сообщения о проблеме или запроса на изменение обычно требуется утверждение службы гарантии качества ПО.

Задача 11: отслеживание и оценка сообщений о проблемах. Служба гарантии качества ПО отслеживает сообщения о проблемах на протяжении работ по разработке и верификации ПО, чтобы убедиться, что соблюдаются процессы регистрации сообщений о проблемах, определенные в Плане управления конфигурацией ПО (Software Configuration Management Plan, SCMP). Сообщения о проблемах также оцениваются на полноту, точность и достаточность. Служба гарантии качества ПО также следит за тем, чтобы все изменения, вызванные сообщениями о проблемах или запросами на изменение, были верифицированы до их закрытия.

Задача 12: оценка процессов и записей УК ПО. Служба гарантии качества ПО проводит аудит записей и процессов управления конфигурацией ПО (Software Configuration Management, SCM), чтобы удостовериться, что они соответствуют плану управления конфигурацией ПО. Служба гарантии качества ПО оценивает управление конфигурацией ПО в ходе разработки и верификации. Служба гарантии качества ПО следит за тем, чтобы данные в системе управления конфигурацией соответствовали данным, указанным в указателе конфигурации ПО; то есть проверяет полноту и точность библиотеки управления конфигурацией и указателя конфигурации ПО. Служба гарантии качества ПО также проводит аудит процесса контроля изменений на протяжении разработки и верификации ПО, чтобы убедиться, что проблемы надлежащим образом документируются, изменения согласовываются, корректно реализуются и верифицируются, а вся документация должным образом обновляется.

Задача 13: аудит поставщиков. Если в проекте используются поставщики или субподрядчики, служба гарантии качества ПО проводит аудит процессов гарантии качества ПО у поставщиков, а также их общих работ по разработке и верификации.

Задача 14: определение корректирующих действий. Выполняя свои задачи, служба гарантии качества ПО может выявлять необходимые корректирующие действия. Эти действия могут документироваться в сообщении о проблеме (Problem Report, PR) или в отдельной записи службы гарантии качества ПО, например в отчете о корректирующем действии. Помимо определения корректирующих действий, служба гарантии качества ПО должна отслеживать их выполнение, чтобы убедиться, что действия действительно реализованы надлежащим образом. Корректирующие действия должны быть завершены до выпуска итогового отчета разработки ПО.

Задача 15: рассмотрение и утверждение отклонений и освобождений. Служба гарантии качества ПО рассматривает любые отклонения или освобождения от утвержденных процессов, чтобы убедиться, что соответствие DO-178C и безопасность не затрагиваются. Служба гарантии качества ПО также следит за тем, чтобы все такие отклонения или освобождения были документированы в соответствии с утвержденным процессом, например в сообщении о проблеме. Аналогичным образом служба гарантии качества ПО рассматривает и утверждает любые исправления по месту, внесенные в утвержденные тестовые процедуры.

Задача 16: проведение рассмотрения соответствия. Как правило, последним шагом работ по обеспечению соответствия DO-178C является рассмотрение соответствия. Согласно разделу 8.3 DO-178C: «Цель рассмотрения соответствия программного обеспечения состоит в том, чтобы получить уверенность, что для программного продукта, представляемого в составе заявки на сертификацию, процессы ЖЦ ПО завершены, данные ЖЦ ПО завершены, а

исполняемый объектный код находится под управлением и может быть регенерирован» [7]. Такое рассмотрение обычно документируется в отчете о рассмотрении соответствия ПО и подтверждает следующее [7]:

- Запланированные процессы ЖЦ ПО завершены, и данные ЖЦ ПО сформированы.
- Данные жизненного цикла трассируются к системным требованиям и требованиям безопасности, из которых они получены.
- Имеются свидетельства того, что данные ЖЦ ПО были созданы и находились под управлением в соответствии с планами и стандартами.
- Имеются свидетельства того, что каждое сообщение о проблеме было оценено и имеет зарегистрированный статус.
- Любые отклонения от требований задокументированы и утверждены, как правило, с использованием процесса работы с сообщениями о проблемах.
- Исполняемый объектный код может быть создан из исходного кода, документированного в указателе конфигурации ПО, с использованием указанных инструкций по сборке. Утвержденное ПО может быть загружено с использованием выпущенных инструкций по загрузке. Любые сообщения о проблемах, отложенные по итогам предыдущих рассмотрений соответствия, переоцениваются.
- Если для зачета в сертификации используется ранее разработанное программное обеспечение, подтверждается, что текущая базовая версия ПО трассируется к предыдущей базовой версии и все изменения утверждены.

Задача 17: документирование мероприятий в записях гарантии качества ПО. Мероприятия по гарантии качества ПО документируются в соответствующих записях, чтобы предоставить свидетельства участия службы гарантии качества ПО на протяжении работ по разработке и верификации ПО. Записи гарантии качества ПО обычно включают следующую информацию: что оценивалось, дата, имя оценщика, критерии оценки, статус оценки, то есть соответствие или несоответствие, замечания, лицо или лица, которые должны ответить, срок ответа, серьезность замечаний и обновления, относящиеся к принятию ответа [10]. Поскольку такие записи считаются данными категории контроля 2 (КК2), им требуется уникальная конфигурационная идентификация. См. главу 10 для информации о КК2. Конфигурационная идентификация обычно обеспечивается включением даты и темы в заголовки файла записи гарантии качества ПО.

Ссылки

1. *The American Heritage Dictionary of the English Language*, 4-е изд. (Boston, MA: Houghton Mifflin Company, 2009).

2. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 7-е изд. (New York: McGraw-Hill, 2010).
3. E. R. Baker and M. J. Fisher, Chapter 1 – Organizing for quality management, в *Software Quality Assurance Handbook*, 4-е изд., G. Gordon Schulmeyer, Ed. (Norwood, MA: Artech House, 2008), pp. 1-34.
4. ISO/IEC 9126, *Software Engineering – Product Quality Int'l Standard* (международный стандарт, 2003).
5. H. Van Vliet, *Software Engineering: Principles and Practice*, 3-е изд. (Chichester, U.K.: Wiley, 2008).
6. W. E. Lewis, *Software Testing and Continuous Quality Improvement*, 2-е изд. (Boca Raton, FL: CRC Press, 2005).
7. DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
8. G. Gordon Schulmeyer, Chapter 2 – Software quality lessons learned from the quality experts, в *Software Quality Assurance Handbook*, 4-е изд., G. Gordon Schulmeyer, Ed. (Norwood, MA: Artech House, 2008), pp. 35-62.
9. Software Engineering Standards Committee of the IEEE Computer Society, *IEEE Standard for Software Quality Assurance Plans*, IEEE Std 730-2002 (New York: IEEE, 2002).
10. J. Meagher and G. Gordon Schulmeyer, Chapter 13 – Development quality assurance, в *Software Quality Assurance Handbook*, 4-е изд., G. Gordon Schulmeyer, Ed. (Norwood, MA: Artech House, 2008), pp. 311-330.

*ISO-9000 называется *Quality Management Systems – Fundamentals and Vocabulary*; ISO-9001 – *Quality Management Systems – Requirements*; а ISO-9004 – *Managing for the Sustained Success of an Organization – A Quality Management Approach*.

*Это указано в таблице А-9 DO-178С, цель 1. Эта цель была частью таблицы А-1 DO-178В, но в DO-178С была перенесена в таблицу А-9. Ответственность гарантии качества ПО за анализ планов на согласованность и соответствие целям DO-178В, а теперь и DO-178С, существовала всегда.

*Согласно таблице А-9 DO-178С, целям 2 и 3, одна из целей гарантии качества ПО состоит в обеспечении соответствия планам и стандартам. Аудит является одним из способов реализации этой цели.

†Часто в плане гарантии качества ПО указывается целевой процент коллегиальных рассмотрений, в которых служба гарантии качества ПО должна участвовать. Этот процент может

определяться общим риском проекта.

Глава 12. Взаимодействие по вопросам сертификации

Сокращения

Сокращение	Расшифровка
CAST	Certification Authorities Software Team (группа сертифицирующих органов по программному обеспечению)
KK1	Категория контроля 1 (control category #1)
KK2	Категория контроля 2 (control category #2)
CM	Configuration management (управление конфигурацией)
CRI	Certification review item (элемент сертификационного рассмотрения)
DER	Designated Engineering Representative (назначенный инженерный представитель)
EASA	European Aviation Safety Agency (Европейское агентство по авиационной безопасности)
FAA	Federal Aviation Administration (Федеральное управление гражданской авиации США)
ODA	Organization Designation Authorization (организационные полномочия по назначению представителей)
PR	Сообщение о проблеме (problem report)
PSAC	Plan for Software Aspects of Certification (План сертификации ПО)
PSCP	Project-Specific Certification Plan (план сертификации для конкретного проекта)
SAS	Software Accomplishment Summary (Итоговый отчёт разработки ПО)
SCI	Software Configuration Index (Указатель конфигурации ПО)
SCM	Software configuration management (управление конфигурацией ПО)
SCMP	Software Configuration Management Plan (План управления конфигурацией ПО)
SDP	Software Development Plan (План разработки ПО)
SECI	Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index)

Сокращение	Расшифровка
SOI	Stage of involvement (этап взаимодействия с сертифицирующим органом)
SQA	Software quality assurance (гарантия качества ПО)
SQAP	Software Quality Assurance Plan (План гарантии качества ПО)
SVP	Software Verification Plan (План верификации ПО)
TQL	Tool qualification level (уровень квалификации инструмента)
TSO	Technical Standard Order (технический стандартный приказ)
WCET	Worst-case execution time (наихудшее время выполнения)

12.1 Что такое взаимодействие с сертифицирующим органом?

Взаимодействие с сертифицирующим органом – это постоянное общение и координация между заявителем[*] и сертифицирующим органом. Это необходимый процесс для успешной сертификации. В большинстве отраслей существует тот или иной процесс сертификации или одобрения. В авиации сертификация – трудоемкий процесс, требующий постоянного внимания. Если требования сертификации не учитываются на протяжении всего проекта, результатом становится либо невозможность получить сертификат, либо задержка сертификации и чрезмерные затраты. В этой главе речь идет о процессе сертификации воздушного судна в части программного обеспечения, как его требуют Федеральное управление гражданской авиации США (FAA) и другие сертифицирующие органы гражданской авиации. DO-178C (КТ-178С) определяет взаимодействие с сертифицирующим органом как *интегральный процесс*, то есть процесс, применимый на протяжении всего ЖЦ ПО. Такое взаимодействие начинается рано, еще на этапе планирования, и завершается окончательным подтверждением соответствия. В таблице А-10 DO-178С указаны три цели процесса взаимодействия с сертифицирующим органом, применимые ко всем уровням ПО. Ниже кратко приведены эти цели [1]:

- *Цель 1 таблицы А-10 DO-178С:* «Установлены коммуникация и взаимопонимание между заявителем и сертифицирующим органом». Это достигается разработкой и представлением плана сертификации ПО (см. главу 5). На большинстве проектов коммуникация между сертифицирующим органом и заявителем продолжается на протяжении всего жизненного цикла проекта.
- *Цель 2 таблицы А-10 DO-178С:* «Предлагаются методы определения соответствия и достигается согласие по Плану сертификации ПО». Это достигается одобрением плана сертификации ПО со стороны сертифицирующего органа, которое обычно сообщается в письме.

- *Цель 3 таблицы A-10 DO-178C:* «Предоставляется подтверждение соответствия». Выходными данными по этой цели являются указатель конфигурации ПО (см. главу 10) и итоговый отчёт разработки ПО (Software Accomplishment Summary, SAS), который описан далее в этой главе. Одобрение этих двух документов означает завершение работ по DO-178C, по крайней мере для данной версии программного обеспечения.

12.2 Взаимодействие с сертифицирующими органами

В большинстве компаний есть четко определенный процесс взаимодействия с сертифицирующим органом: может быть отдел сертификации (иногда его называют отделом летной годности), могут привлекаться представители, уполномоченные сертифицирующим органом, например назначенные инженерные представители (Designated Engineering Representatives, DER) или участники подразделения организационных полномочий по назначению представителей (Organization Designation Authorization, ODA), либо применяется сочетание этих вариантов. Как правило, отдел сертификации занимается организационной координацией с Федеральным управлением гражданской авиации США (Federal Aviation Administration, FAA) и базовыми сертификационными работами, а назначенные представители и участники такого подразделения сосредоточены на технической стороне подтверждения соответствия нормативным требованиям и руководящим материалам по сертификации. В крупных компаниях также могут быть инженеры по взаимодействию с сертификацией, которые готовят данные и координируют с назначенными представителями работы по подтверждению соответствия DO-178C. Конкретная организация процесса взаимодействия по вопросам сертификации зависит от местонахождения заявителя и сертифицирующего органа, типа запрашиваемого одобрения (например, сертификация типа воздушного судна, двигателя или воздушного винта либо авторизация изделия по техническому стандартному приказу (Technical Standard Order, TSO)), отношения разработчика программного обеспечения к заявителю и других факторов. Дополнительная информация о процессе сертификации Федерального управления гражданской авиации США приведена в приказе FAA Order 8110.4 *Type Certification* и/или приказе 8150.1 *Technical Standard Order Program*.[*]

Как отмечалось в главе 5, важно добиться согласия сертифицирующего органа по плану сертификации ПО. Взаимодействие с сертифицирующим органом начинается еще до представления этого плана. Регулярные технические совещания от начала до конца проекта являются эффективным средством координации. Совещания не заменяют необходимость в данных, но предоставляют площадку для обсуждения и разрешения вопросов на протяжении всего проекта с целью, чтобы ни для одной из сторон не было *сюрпризов*. Чтобы фиксировать взаимодействие с сертифицирующим органом и контролировать выполнение согласованных действий, рекомендуется вести подробные протоколы совещаний и посто-

янно поддерживать перечень открытых действий.

Сертифицирующий орган или назначенный представитель обычно выполняет оценку уровня участия в начале проекта, формально или неформально. Эта оценка учитывает опыт команды в сертификации и в подтверждении соответствия DO-178C (или DO-178B), новизну и необычность предлагаемой технологии, критичность программного обеспечения для функциональности воздушного судна, репутацию команды по ПО и авионике или электрическим системам, качество персонала, отвечающего за взаимодействие по сертификации (например, назначенных представителей), и так далее. Приказ FAA Order 8110.49, глава 3 и приложение 1, позволяют понять процесс и критерии оценки, применяемые сертифицирующим органом [2].[†] Для проектов, которые оценены как высокорисковые и потенциально влияющие на безопасность, уровень участия сертифицирующего органа будет высоким. Для проектов с меньшим риском и критичностью сертифицирующие органы могут делегировать значительную часть работ по подтверждению соответствия назначенному представителю. Оценка уровня участия определяет следующее:

- Какой объем данных необходимо представлять в сертифицирующий орган. Для проектов более высокого риска часто требуется представлять все планы и результаты испытаний в дополнение к плану сертификации ПО, указателю конфигурации ПО и итоговому отчету разработки ПО.
- Сколько аудитов будет выполнять сертифицирующий орган.
- Какая часть работ по подтверждению соответствия будет делегирована назначенным представителям.
- Насколько часто заявитель и сертифицирующий орган должны будут проводить встречи.

12.2.1 Рекомендуемые практики координации с сертифицирующими органами

Как отмечалось ранее, подход к взаимодействию между заявителем и сертифицирующим органом различается в зависимости от особенностей проекта и предпочтений органа. Большая часть выводов о соответствии обычно делегируется назначенным представителям или уполномоченной организации, при этом сертифицирующий орган осуществляет надзор. Ниже приведены некоторые рекомендации по эффективной координации с сертифицирующими органами.

Рекомендация 1: Представляйте проектный план. Многие организации представляют план сертификации для конкретного проекта (Project-Specific Certification Plan, PSCP) или эквивалентный документ, в котором описываются общие планы проекта. Такой план включает описание проекта, в том числе воздушного судна и систем, определяет планируемые рабо-

ты по программному обеспечению и электронному аппаратному обеспечению воздушного судна, с указанием уровней программного и аппаратного обеспечения и выбранных поставщиков или разработчиков, определяет предлагаемые методы определения соответствия применимым нормативным требованиям, указывает контактных лиц, содержит график проекта и так далее. Такой план задает рамки для работ по ПО и инициирует взаимодействие с сертифицирующим органом.

Рекомендация 2: Проводите раннее ознакомительное совещание. Чем раньше сертифицирующий орган познакомится с проектом, тем лучше. Первое ознакомительное совещание обычно назначается примерно в то же время, когда этот план готов к представлению. Стартовое совещание дает общее представление о проекте, системе, программном обеспечении, графике и планах сертификации. Хотя основная цель такого совещания состоит в ознакомлении сертифицирующего органа с планируемым проектом, оно также дает ценную информацию заявителю. Сертифицирующий орган обычно указывает на проблемные области и ожидаемые вопросы сертификации, обсуждает предполагаемые роли и обязанности, разъясняет ожидания по взаимодействию и предлагает следующие шаги.

Рекомендация 3: Выявляйте потенциальные сертификационные и технические проблемы. Следует прямо сообщать о любых ожидаемых проблемах сертификации или технических сложностях. Важно информировать органы, если планируется применение новой технологии или нетрадиционных подходов. В целом сертифицирующие органы не любят сюрпризов. Сюрпризы могут создать впечатление, что кто-то что-то скрывает, и снизить уровень доверия со стороны сертифицирующего органа. Поэтому важно честно говорить о потенциальных сложностях. Не менее важно определить планы по снижению рисков или устранению этих сложностей. Раннее раскрытие потенциальных проблем позволяет органам официально зафиксировать свои опасения. Федеральное управление гражданской авиации США выпускает проблемные записки по сертификации для выявления уникальных вопросов подтверждения соответствия. Затем заявитель предоставляет ответы на эти проблемные записки, чтобы объяснить предполагаемый подход к работе с проблемой. У других сертифицирующих органов существуют аналогичные механизмы выявления сертификационных вопросов. Например, Европейское агентство по авиационной безопасности (European Aviation Safety Agency, EASA) использует элементы сертификационного рассмотрения (certification review items, CRI), а сертифицирующий орган Канады использует сертификационные меморандумы.

Рекомендация 4: Своевременно отвечайте на проблемные записки. Хотя проблемные записки трудно назвать приятными, чем раньше они выпущены и закрыты, тем лучше. Получение такой записки на позднем этапе программы может крайне дестабилизировать проект. Как только она получена, важно как можно быстрее подготовить полный ответ. Большинство

таких записок можно предвидеть заранее. Федеральное управление гражданской авиации США располагает стандартным набором проблемных записок по программному обеспечению, которые выпускаются почти на всех новых сертификационных проектах. Примерами распространенных тем для таких записок являются объектно-ориентированные технологии, модельно-ориентированная разработка и покрытие объектного кода.[*] Эти общие проблемные записки иногда называют типовыми проблемными записками, и они служат отправной точкой для записок, относящихся к конкретному проекту. Кроме того, могут существовать и другие проблемные записки, специфичные для проекта, в зависимости от его сложностей. Например, если значительная часть работ передана внешним подрядчикам или вынесена в другие страны, может появиться проблемная записка по вопросам надзора и регистрации сообщений о проблемах. Ответ заявителя на каждую такую записку должен прояснять все спорные моменты, поскольку иногда сама записка может содержать не вполне точное понимание деталей, и должен объяснять, как соответствующий вопрос будет решаться в конкретном проекте. Важно продолжать взаимодействие с сертифицирующим органом до тех пор, пока проблемная записка не будет закрыта, то есть пока не будет получено одобрение позиции проекта со стороны сертифицирующего органа. Иногда для достижения согласия требуется несколько итераций. Неурегулированная проблемная записка может быть столь же рискованной, как и та, которая еще даже не была выпущена.

Рекомендация 5: Готовьте, предварительно представляйте и подавайте план сертификации ПО заранее. Как отмечалось в главе 5, лучше подготовить и представить план сертификации ПО на раннем этапе. Чем дольше проект откладывает представление плана сертификации ПО, тем выше риск в случае, если сертифицирующий орган отклонит план. Для проектов высокого риска настоятельно рекомендуется предварительно представить план сертификации ПО сертифицирующему органу до его официальной подачи. Это позволяет получить неформальную обратную связь, при необходимости скорректировать план или, напротив, убедиться, что его можно подавать. Подробности по Плану сертификации ПО (PSAC) приведены в главе 5.

Рекомендация 6: Быстро реагируйте на любые вопросы или замечания по Плану сертификации ПО (PSAC). У сертифицирующего органа часто возникают вопросы по Плану сертификации ПО (PSAC). Важно отвечать на них быстро, полно и точно. Сертифицирующие органы обычно перегружены работой и не избалованы ресурсами, поэтому, когда они рассматривают ваши данные, лучше отвечать без задержек. Иначе принятие плана сертификации ПО может затянуться.

Иногда сертифицирующий орган просит внести некоторые обновления в план сертификации ПО. Важно правильно понять суть замечаний; если требуется уточнение, может помочь совещание или телефонный звонок. Нередко вопросы снимаются в ходе короткого обсуж-

дения и, возможно, соглашения сделать соответствующую пометку в итоговом отчёте разработки ПО. Такие договоренности следует фиксировать письменно; обычно для этого достаточно электронного письма или протокола совещания. Если после обсуждения вопросы остаются, представьте предполагаемые изменения сертифицирующему органу и получите общее согласие до внесения обновлений в план сертификации ПО и его повторного представления.

Рекомендация 7: Встречайтесь с сертифицирующими органами на протяжении всего проекта. Для проектов, требующих высокого уровня участия сертифицирующего органа, следует организовать периодические встречи с ним. Частота таких встреч может меняться по ходу проекта в зависимости от развития ситуации. Бывают периоды, когда необходимы ежемесячные встречи. В другие моменты могут быть более уместны квартальные встречи или встречи по мере необходимости. Для проектов высокого риска с назначенными представителями следует консультироваться на протяжении всего проекта и привлекать их ко всей координации с сертифицирующими органами.

Если проект требует среднего или низкого уровня участия сертифицирующего органа, большая часть координации, вероятно, будет вестись с назначенными представителями или с корпоративной группой взаимодействия по сертификации.

Взаимодействие с сертифицирующими органами должно быть сосредоточено на состоянии проекта, возникших проблемах и плане их решения. Следует определять действия и сроки их выполнения и включать их в официальный протокол совещания. Важно выполнять согласованные действия.

Рекомендация 8: Решайте проблемы по мере их возникновения. Проблемы редко исчезают сами собой, если их игнорировать. Их следует оперативно устранять до того, как ситуация выйдет из-под контроля. Обязательно информируйте назначенных представителей или персонал по взаимодействию с сертификацией о любых вопросах, которые могут повлиять на соответствие. Они помогут определить предпочтительный способ информирования сертифицирующих органов. При сообщении о проблемах органам сертификации всегда следует иметь предложенный план их решения.

Рекомендация 9: Представляйте любые изменения или отклонения сертифицирующему органу для одобрения или принятия. Как отмечалось в главе 5, план сертификации ПО должен определять план работы с изменениями в утвержденных процессах. Сертифицирующие органы, скорее всего, захотят быть информированными о любых изменениях процессов на протяжении ЖЦ ПО.

Рекомендация 10: Поддерживайте аудиты, проводимые сертифицирующими органами и/или их назначенными представителями. Проекты по программному обеспечению обычно

проходят аудит со стороны сертифицирующих органов или их назначенных представителей для подтверждения соответствия DO-178C и выявленным проблемным запискам или их эквивалентам. Аудиты требуют подготовки и сопровождения. Дополнительная информация о том, чего ожидать во время аудита и как к нему готовиться, приведена далее в этой главе.

Рекомендация 11: Своевременно представляйте данные по подтверждению соответствия программного обеспечения. Указатель конфигурации ПО и итоговый отчёт разработки ПО являются двумя элементами данных жизненного цикла, которые обычно представляются в сертифицирующий орган для демонстрации соответствия DO-178C и нормативным требованиям. Эти материалы обычно подаются за несколько месяцев до окончательной выдачи сертификата типа на воздушное судно или двигатель. Для проектов по авторизации по техническому стандартному приказу указатель конфигурации ПО и итоговый отчёт разработки ПО представляются вместе с пакетом авторизации. Следует заранее координировать представление данных, чтобы у сертифицирующего органа было достаточно времени на их рассмотрение и одобрение.

В некоторых проектах сертифицирующий орган может также потребовать представления результатов верификации программного обеспечения в составе данных по подтверждению соответствия. Ожидаемый состав представляемых материалов обычно обсуждается с сертифицирующим органом на этапе планирования и документируется в плане сертификации ПО.

12.3 Итоговый отчёт разработки ПО

В таблице A-10 DO-178C указаны три вида данных, поддерживающих процесс взаимодействия по вопросам сертификации: план сертификации ПО, указатель конфигурации ПО и итоговый отчёт разработки ПО. Как отмечалось ранее, иногда также представляются результаты верификации программного обеспечения. Содержимое плана сертификации ПО описано в главе 5; указатель конфигурации ПО описан в главе 10; результаты верификации ПО описаны в главе 9. Далее описано ожидаемое содержимое итогового отчёта разработки ПО.

Итоговый отчёт разработки ПО суммирует работы по подтверждению соответствия DO-178C. Раздел 11.20 DO-178C дает рекомендации по его содержанию. План сертификации ПО и итоговый отчёт разработки ПО образуют пару документов: план подается в начале проекта и объясняет, что будет сделано, а итоговый отчёт подается в конце проекта и объясняет, что фактически было сделано. Итоговый отчёт разработки ПО содержит во многом тот же материал, что и план сертификации ПО, включая обзор системы, обзор программного обеспечения, соображения по сертификации, сводку по жизненному циклу, сводку по данным жизненного цикла или ссылку на них, но итоговый отчёт пишется в прошедшем

времени, а не в будущем. Кроме того, итоговый отчёт разработки ПО содержит следующее [1]:

1. *Отклонения* от утвержденных планов и процессов. Это часто отмечается по всему тексту итогового отчёта разработки ПО и суммируется в приложении. Например, если фактически использованные инструменты отличались от запланированных, это должно быть объяснено.
2. *Конфигурацию* программного обеспечения, подлежащего одобрению.
3. *Открытые или отложенные сообщения о проблемах*, включая обоснование того, почему они не влияют на безопасность, функциональность, эксплуатацию или соответствие требованиям. Обычно включается таблица, в которой сведена следующая информация по всем сообщениям о проблемах, не закрытым на момент сертификации или авторизации по техническому стандартному приказу [1 – 3]:

- a. Номер сообщения о проблеме
- b. Заголовок сообщения о проблеме
- c. Дата открытия
- d. Описание проблемы, включая первопричину, если она известна, и затронутые данные
- e. Классификация (обсуждение классификаций сообщений о проблемах см. в главе 10)
- f. Обоснование отсрочки, включая объяснение, почему проблема не оказывает отрицательного влияния на безопасность, функциональность, эксплуатацию или соответствие нормативным требованиям, включая XX.1301 и XX.1309[*]
- g. Средства снижения риска, если применимо, например, эксплуатационные ограничения или функциональные ограничения
- h. Связь с другими открытыми сообщениями о проблемах

Кроме того, некоторые сертифицирующие органы могут потребовать включать в итоговый отчёт разработки ПО план закрытия сообщений о проблемах. Это относительно новое ожидание со стороны сертифицирующих органов; оно появилось для того, чтобы стимулировать закрытие сообщений о проблемах, а не оставлять их открытыми годами.

Также хорошей практикой является пояснить в итоговом отчёте разработки ПО, каким образом любые проблемы, выявленные после сертификации, будут документироваться, оцениваться и управляться.

4. *Закрытые PR*: если итоговый отчёт разработки ПО готовится для последующей сертификации или основан на ранее разработанном программном обеспечении, должны быть указаны все сообщения о проблемах или запросы на изменение, возникшие с момента последнего одобрения (либо в итоговом отчёте разработки ПО, либо в указателе конфигурации ПО).
5. *Характеристики программного обеспечения*, такие как размер исполняемого объектного кода, запасы по времени и запасы по памяти. Анализы для определения этих характеристик описаны в главе 9.
6. *Заявление о соответствии*, в котором говорится, что программное обеспечение соответствует DO-178C и любым другим требованиям сертификации. Часто итоговый отчёт разработки ПО включает матрицу соответствия или ссылается на нее; в этой матрице кратко показано, как удовлетворяется каждая цель DO-178C и какие данные подтверждают соответствие.

12.4 Аудиты этапов взаимодействия с сертифицирующим органом

12.4.1 Обзор аудитов этапов взаимодействия

В середине 1990-х годов сертифицирующие органы как в Соединенных Штатах, так и в Европе начали проводить аудиты программных проектов для оценки соответствия DO-178B. К сожалению, выяснилось, что многие проекты не выполняли цели стандарта. Также было замечено, что подходы сертифицирующих органов к оценке проектов сильно различались. Из-за этого международная группа сертифицирующих органов по программному обеспечению (Certification Authorities Software Team, CAST) скоординировала работу и определила подход к оценке соответствия ПО. В документе этой группы были выделены четыре контрольные точки на протяжении проекта: (1) планирование, (2) разработка, (3) испытания и (4) финальная оценка. Федеральное управление гражданской авиации США дополнительно задокументировало этот подход в главе 2 приказа Order 8110.49 и в пособии Software Review Job Aid под названием *Conducting Software Reviews prior to Certification*.[*] В приказе Order 8110.49 объясняется, что именно будут делать Федеральное управление гражданской авиации США или уполномоченные представители и какие данные будут проверяться.[†] Пособие описывает, как сертифицирующие органы будут выполнять *рассмотрения* (также называемые рассмотрениями этапов взаимодействия с сертифицирующим органом (Stage of Involvement, SOI)). Изначально это пособие задумывалось как обучающий инструмент для инженеров Федерального управления гражданской авиации США и их представителей, чтобы стандартизировать процесс рассмотрения. Однако теперь для многих проектов FAA требует, чтобы вопросы из пособия были заполнены заявителем или его представителями до представления сертификационных данных. В таблице 12.1 дан обзор номера и типа этапа

взаимодействия, проверяемых данных и оцениваемых целей DO-178C.

В пособии и приказе FAA Order 8110.49 используется термин «рассмотрения этапов взаимодействия»; однако термин *аудит* позволяет отличать их от рассмотрений верификации, поэтому в этой главе используются термины *аудит*, *аудитор* и *аудируемая сторона*. [‡] Ниже будет приведена дополнительная информация о проведении аудита этапа взаимодействия (для аудиторов) и подготовке к такому аудиту (для аудируемой стороны).

12.4.2 Обзор документа Software Review Job Aid

Разделы 9.2, 10.3 DO-178C поясняют, что сертифицирующий орган может выполнять рассмотрения, однако не дают дальнейших разъяснений о том, что именно включают эти рассмотрения. Приказ Order 8110.49 и пособие FAA Software Review Job Aid дают дополнительное объяснение того, *что* будет оцениваться (Order 8110.49) и *как* это оценивать (пособие). В таблице 12.1 приведен обзор четырех аудитов этапов взаимодействия, включая данные и цели DO-178C, которые оцениваются. Пособие содержит рекомендации по проведению аудита этапа взаимодействия, а также перечисляет действия и вопросы, подлежащие оценке во время аудита. Пособие дает представление о логике и ожиданиях сертифицирующих органов; это понимание помогает разработчикам готовиться к аудитам и лучше планировать проекты с самого начала. Некоторые компании используют это пособие для проведения самоаудитов.

Таблица 12.1 Краткое описание четырех этапов взаимодействия

Номер и тип этапа взаимодействия	Проверяемые данные	Оцениваемые цели DO-178C
1. Планирование	План сертификации ПО, План разработки ПО (SDP), План верификации ПО (SVP), План управления конфигурацией ПО (SCMP), План гарантии качества ПО (SQAP), стандарты разработки, результаты верификации (записи рассмотрений планирования), записи гарантии качества ПО, записи управления конфигурацией ПО и планы квалификации инструментов (если они оформлены отдельно от плана сертификации ПО).	Преимущественно таблица A-1; таблицы A-8 по A-10 в той части, в какой они относятся к планированию.

Номер и тип этапа взаимо- действия	Проверяемые данные	Оцениваемые цели DO-178C
2. Разработка	Любые данные планирования, не завершенные на этапе взаимодействия 1 или изменившиеся после него, системные требования, распределенные на ПО, требования к ПО (высокого уровня и производные высокого уровня), Описание проекта ПО (требования низкого уровня, производные требования низкого уровня, архитектура), исходный код, процедуры сборки, Указатель конфигурации среды ЖЦ ПО, записи верификации (для требований, проекта и кода), данные трассировки, сообщения о проблемах и/или запросы на изменение, записи гарантии качества ПО и записи управления конфигурацией ПО.	Преимущественно таблицы А-2 по А-5; таблицы А-8 по А-10 в той части, в какой они относятся к разработке.
3. Верификация (тесты)	Данные, не проверенные или не закрытые на предыдущих этапах взаимодействия, объектный код, тестовые примеры и процедуры верификации ПО, результаты верификации, Указатель конфигурации среды ЖЦ ПО, Указатель конфигурации ПО с тестовой базовой версией, данные трассировки, сообщения о проблемах и/или запросы на изменение, записи гарантии качества ПО и записи управления конфигурацией ПО.	Преимущественно таблицы А-6, А-7; таблицы А-8 по А-10 в той части, в какой они относятся к тестам.

Номер и тип этапа взаимодействия	Проверяемые данные	Оцениваемые цели DO-178C
4. Финальный	Любые данные, не завершенные или не проверенные на предыдущих этапах взаимодействия, результаты верификации (часто оформляемые как результаты верификации ПО), Указатель конфигурации среды ЖЦ ПО, Указатель конфигурации ПО, Итоговый отчёт разработки ПО, сообщения о проблемах и/или запросы на изменение, данные трассировки, записи гарантии качества ПО, отчёт о рассмотрении соответствия ПО и записи управления конфигурацией ПО.	Преимущественно таблица А-10; цели, которые не были оценены на предыдущих этапах взаимодействия; цели, по которым были отмечены проблемы на предыдущих этапах взаимодействия.

Пособие разделено на четыре части. Часть 1 дает общий обзор, включая введение в четыре аудита этапов взаимодействия и определение некоторых ключевых терминов. Ниже приведены термины, которые особенно важно понимать [4]:[*]

- *Соответствие* – это выполнение цели DO-178B.
- *Наблюдение о несоответствии* (finding) – это выявление факта, что не продемонстрировано соответствие одной или нескольким целям RTCA/DO-178B.
- *Замечание* (observation) – это выявление потенциального улучшения процесса ЖЦ ПО. Замечание не является вопросом соответствия RTCA/DO-178B и не требует устранения до одобрения программного обеспечения.
- *Действие* (action) – это задание организации или лицу с датой завершения для исправления finding, ошибки или недостатка, выявленного при проведении рассмотрения программного обеспечения.
- *Вопрос* (issue) – это проблема, не относящаяся непосредственно к соответствию программного обеспечения или улучшению процесса, но представляющая собой предмет беспокойства в области безопасности, системной инженерии, управления программой, организации или иной области, обнаруженный во время рассмотрения ПО.

Часть 2 объясняет типовые задачи аудита. Независимо от типа этапа взаимодействия (пла-

нирование, разработка, верификация/тестирование или финальный этап), аудитор должен подготовиться к аудиту, провести аудит, задокументировать результаты аудита, подвести итоги аудита на заключительном совещании (или иногда в письменной краткой сводке для руководства) и выполнить последующие

действия (например, подготовить отчет по этапу взаимодействия и убедиться, что все выявленные несоответствия, замечания и действия были обработаны).

Часть 3 составляет основную часть пособия. В ней суммируются действия и вопросы для каждого аудита этапа взаимодействия. Для каждого такого аудита приводится таблица, где перечислены действия этапа взаимодействия (выделенные жирным шрифтом) и вопросы, используемые для выполнения каждого действия. Каждый вопрос сопоставлен с целью(ями) DO-178B, которые он оценивает.[*] В таблице 12.2 приведен пример одного действия этапа взаимодействия 1 и нескольких поддерживающих вопросов. Обычно во время аудита этапа взаимодействия в таблицу добавляют столбец, в котором приводится ответ на каждый вопрос на основе оценки данных проекта, и эта таблица включается в отчет по аудиту этапа взаимодействия.

Таблица 12.2 Фрагмент пособия Software Review Job Aid: действия и вопросы этапа взаимодействия 1

№ пункта	Действие или вопрос оценки этапа взаимодействия 1	
	Цель(и) DO-178B	
1.1	Проверить все планы, а именно план сертификации ПО, план управления конфигурацией ПО, план гарантии качества ПО, План разработки ПО (SDP), план верификации ПО, планы квалификации программного инструмента и стандарты. По результатам рассмотрения всех планов рассмотреть следующие вопросы.	
	А-1, пункты 1-7	

№ пункта	Действие или вопрос оценки этапа взаимодействия 1	Цель(и) DO-178B
1.1.1	Подписаны ли плановые данные и поставлены ли они под управление конфигурацией, то есть КК1 или КК2 в зависимости от уровня ПО? Проверить наличие объективных свидетельств координации, например разрешенных подписей, со стороны всех организаций, контролирующих и затрагиваемых программными планами и стандартами.	А-1, пункты 1-7
1.1.2	Были ли планы и стандарты рассмотрены и утверждены уполномоченным специалистом, если это требуется?	А-1, пункты 1-7
1.1.3	Являются ли планы и стандарты полными, понятными и согласованными?	А-1, пункты 1 и 7
1.1.4	Проводились ли рассмотрения планов и стандартов, сохранялись ли результаты рассмотрений, и устранялись ли выявленные замечания? См. раздел 4.6 DO-178B.	А-1, пункты 6 и 7

№ пункта	Действие или вопрос оценки этапа взаимодействия 1	Цель(и) DO-178B
1.1.5	Соответствуют ли планы и стандарты содержанию, заданному в разделе 11 DO-178B, то есть разделах 11.1-11.8? Примечание: планы и стандарты не обязаны быть оформлены именно в виде пакета из пунктов 11.1-11.8, однако соответствующие элементы должны быть где-либо документированы.	А-1, пункты 1-7
1.1.6	Описывают ли планы и стандарты процесс внесения изменений в ПО и процедуры для программного обеспечения воздушного судна и процесса применения инструментов, если инструменты используются?	А-1, пункты 1 и 2
1.1.7	Идентифицированы ли в планах все программные инструменты и включено ли обоснование, почему каждый из них требует или не требует квалификации?	А-1, пункт 4
1.1.8	Определены ли для каждого процесса входы, действия, критерии перехода и выходы?	А-1, пункт 1

№ пункта	Действие или вопрос оценки этапа взаимодействия 1	Цель(и) DO-178B
1.1.9	Определены ли в достаточной степени действия жизненного цикла разработки и верификации, с отсылкой к разделам 11.1-11.3 DO-178B, чтобы удовлетворить разделу 4.2 DO-178B?	А-1, пункты 1-7
1.1.10	Соответствуют ли планы и стандарты целям планирования DO-178B из таблицы А-1, то есть являются ли каждый план и стандарт внутренне согласованными, согласованы ли планы и стандарты друг с другом, определен ли жизненный цикл ПО и определены ли критерии перехода?	А-1, пункт 7
1.1.11	Если планы и стандарты будут соблюдаться, обеспечит ли это достижение всех применимых целей DO-178B из таблиц А-2-А-10, то есть описывают ли планы и стандарты, как будут удовлетворяться все применимые цели DO-178B?	А-2-А-10, все применимые цели

Источник: Federal Aviation Administration, *Conducting Software Reviews prior to Certification*, Aircraft Certification Service, Rev. 1, January 2004.

Часть 4 пособия содержит несколько примеров того, как суммировать результаты аудита этапа взаимодействия. Типичное содержимое такого отчета обсуждается далее в этой главе.

Пособие также включает четыре дополнения для помощи аудиторам и проектным командам, проходящим аудит (аудируемой стороне), в том числе следующие:

- Дополнение 1: «Типовые роли и обязанности группы FAA по программному обеспечению». Это дополнение кратко описывает, чего ожидать от различных подразделений FAA в ходе программного проекта. Большинство аудитов этапов взаимодействия выполняются назначенными представителями, но сертифицирующие органы могут участвовать в некоторых высокорисковых проектах.
- Дополнение 2: «Типовые роли и обязанности назначенного специалиста по программному обеспечению». Это дополнение кратко описывает, чем занимается такой назначенный специалист. Эта информация может быть весьма полезной для правильного использования назначенных представителей.
- Дополнение 3: «Примеры писем, повесток и отчета». Это дополнение содержит примеры материалов, которыми аудиторы могут пользоваться при уведомлении заявителей об аудитах. В него также включен пример отчета.
- Дополнение 4: «Необязательные рабочие листы для проверяющих». Это дополнение включает несколько рабочих листов, которые могут помочь аудиторам в ведении записей. Они считаются необязательными и предоставлены главным образом в учебных целях.

12.4.3 Использование пособия *Software Job Aid*

В следующих разделах приведены рекомендации для аудиторов и аудируемой стороны. Сначала даны общие рекомендации, затем кратко описано, чего ожидать во время аудита этапа взаимодействия и как к нему готовиться. Поскольку мне довелось провести или поддержать сотни аудитов, последующие разделы основаны на моем личном опыте и извлеченных уроках. Так как эта информация подается через призму личного опыта, она изложена в более живом тоне, чем другие разделы этой книги.[*] Обратите внимание, что в последующих разделах термин *заявитель/разработчик* используется для обозначения *заявителя и/или разработчика*. Иногда заявитель сам является разработчиком программного обеспечения; однако во многих случаях разработчик ПО является поставщиком для заявителя. Когда заявитель и разработчик являются разными организациями, на аудите должны присутствовать обе стороны.

12.4.4 Общие рекомендации для аудитора

Ниже приведены некоторые рекомендации для тех, кто оказывается в роли аудитора. Это может быть назначенный представитель, сертифицирующий орган, инженер по взаимодействию с сертифицирующим органом, инженер по качеству программного обеспечения или инженер проекта.

Независимо от того, по какой причине вам требуется проводить аудит этапа взаимодействия, эти рекомендации предназначены для того, чтобы помочь вам работать лучше и избегать распространенных ошибок.

Рекомендация аудитору 1: заранее сообщайте и согласовывайте план аудита. За несколько недель до аудита свяжитесь с контактным лицом заявителя/разработчика, чтобы обсудить следующее:

- Убедитесь, что вы понимаете текущее состояние заявителя/разработчика, и подтвердите, что они соответствуют или будут соответствовать согласованным входным критериям для аудита этапа взаимодействия (типовые входные критерии обсуждаются далее).
- Согласуйте дату аудита.
- Подготовьте повестку для выездного или дистанционного аудита[*] (в дополнении 3 к пособию приведены примеры повесток).
- Убедитесь, что ожидания ясно доведены до всех сторон.

Сообщите заявителю/разработчику, кто будет входить в команду аудита, сколько времени займет аудит, какое помещение потребуется для встреч (одна или несколько комнат) и так далее. Чем лучше план аудита доведен заранее, тем более гладко все обычно проходит на месте. Подробное планирование помогает сделать время на площадке более продуктивным и даже более приятным.

Рекомендация аудитору 2: возьмите помощника. При оценке соответствия крайне полезно иметь напарника, особенно для более формальных аудитов (например, аудитов этапов взаимодействия, проводимых в зачет сертификации). У командного подхода есть несколько преимуществ. Во-первых, это занимает меньше времени: команда может разделить задачи, чтобы быстрее изучить большой объем данных. Во-вторых, такой подход обеспечивает более тщательное рассмотрение. Несколько пар глаз и несколько голов быстрее выявляют больше проблем. Я особенно считаю полезной работу в команде для этапов взаимодействия 2 и 3, где приходится анализировать большой объем данных. В-третьих, командная работа дает свидетеля. Мне приходилось участвовать в нескольких аудитах, где проект выдумывал истории о том, что именно происходило. Когда есть свидетель, сделать это сложнее.

Разумеется, замечательно, если ваш напарник является технически опытным участником

рассмотрения. Однако иногда это невозможно из-за ограниченности ресурсов. Все равно полезно иметь человека в роли поддержки, даже если он еще недостаточно опытен, чтобы вести аудит самостоятельно. Более того, это может быть отличным способом подготовки будущих аудиторов. Такой помощник может вести заметки, рассматривать данные по управлению конфигурацией (СМ, управление конфигурацией) и обеспечению качества, изучать записи рассмотрений, присутствовать при сборке и загрузке и так далее.

На одном из моих первых аудитов я приехала одна. Проект был в ужасном состоянии. Прослеживаемости не было; требования, проект и код не совпадали; обеспечение качества фактически отсутствовало. Я провела аудит, сделала заключительный брифинг и уехала в аэропорт. Позже я узнала, что заявитель, который частично сам был виноват в этом беспорядке, поскольку не обеспечил должного надзора, использовал результаты аудита, чтобы жестко надавить на поставщика. Заявитель выборочно слышал заключительный брифинг и использовал его, чтобы возложить всю вину исключительно на поставщика. Замечания аудита были и неверно поняты, и неверно процитированы. При наличии свидетеля заявителю было бы сложнее манипулировать результатами.

С другой стороны, недавно мне пришлось пройти через сложный проект, в котором было слишком много проблем с соответствием. На этот раз у меня было два напарника. Один представлял заявителя, который сам не был разработчиком, а второй был стажером. Во время аудитов мы делили между собой нагрузку. Это позволило изучить больше данных и посмотреть на них под разными углами. Находить так много проблем было неприятно, но оценка получилась гораздо более тщательной. Позже проект оценивали несколько сертифицирующих органов. Поскольку на первых аудитах мы проверили его настолько глубоко, последующие замечания оказались уже совсем незначительными.

Рекомендация аудитору 3: будьте подготовлены. К аудиту необходимо готовиться. Это может включать чтение или повторное чтение планов (если они изменялись или если с момента прошлого аудита прошло много времени), рассмотрение ответов на предыдущие аудиты этапов взаимодействия и закрытие замечаний из предыдущих аудитов этапов взаимодействия. Неподготовленные аудиторы обычно малоэффективны.

Рекомендация аудитору 4: будьте внимательны и доброжелательны. По моему опыту, профессионализм и уважительное отношение гораздо продуктивнее, чем игры во власть и тактика запугивания. Как говорят: мед приносит лучшие результаты, чем уксус. Люди лучше реагируют, когда с ними обращаются уважительно и не заставляют их чувствовать угрозу.

Однажды я работала с человеком, который этому совету не следовал. Он был крайне конфронтационен по отношению к разработчикам. Когда инженеры или программисты прихо-

дили к нему, у них буквально дрожали руки. Когда он задавал вопрос, они не знали, что ответить, и просто смотрели на него. Мы прозвали этого аудитора *Headlight John*, потому что в его присутствии все выглядели как олень в свете фар. Любопытно, что Джон так и не получал от разработчиков действительно точной информации: они были слишком запуганы, чтобы свободно с ним говорить.

Его манера общения сбивала им ход мысли, поэтому они не могли внятно объяснить то, что знали.

Рекомендация аудитору 5: стремитесь по-настоящему понять систему, процесс и реализацию. В первой части аудита заявитель/разработчик даст вам обзор высокого уровня своей системы, программного обеспечения, процессов и текущего состояния. Важно понять общую рамку проекта, процессы и его философию. Прочитайте планы заранее и задавайте вопросы во время презентаций и демонстраций, чтобы убедиться, что вы понимаете, что именно разрабатывается и как именно это разрабатывается. Важно увидеть общую картину, лес, прежде чем уходить в детали, деревья.

Рекомендация аудитору 6: сразу и на всем протяжении аудита этапа взаимодействия сообщайте свои намерения. Каждый день обязательно сообщайте руководителю команды заявителя/разработчика о своих намерениях. Если планы меняются, дайте им об этом знать. Я часто отвожу несколько минут в начале и в конце каждого дня, чтобы объяснить, где мы находимся и куда движемся. Большинство заявителей/разработчиков реагируют вполне конструктивно, если просто понимают, что у вас на уме.

Рекомендация аудитору 7: сначала попросите заявителя/разработчика провести вас по данным. Если эта компания вам незнакома или проект для вас еще сравнительно новый, полезно сначала попросить заявителя/разработчика показать вам свои данные. Пусть они покажут свои требования, проект, код, тестовые примеры, процедуры верификации и результаты верификации. Пусть продемонстрируют, как у них устроена трассируемость. Вы можете попросить показать один нисходящий проход по цепочке (от системного требования к требованию программного обеспечения высокого уровня, далее к требованию низкого уровня и затем к коду) и один восходящий проход (от кода к требованию низкого уровня, затем к требованию высокого уровня и далее к системному требованию). После этого вам, возможно, будет уже комфортно изучать данные самостоятельно, либо вы можете оставить заявителя/разработчика в роли своего водителя, то есть человека, который работает за компьютером по вашим указаниям. Лично я предпочитаю управлять сама, но некоторым аудиторам нравится, когда за клавиатурой работает заявитель/разработчик, а они могут свободно делать заметки и задавать вопросы. Оба варианта допустимы, просто заранее сообщите заявителю/разработчику, что именно вы предпочитаете.

Рекомендация аудитору 8: настойчиво добивайтесь ответов. Иногда, чтобы докопаться до сути проблемы, требуется несколько попыток. Не все проблемы очевидны. По моему опыту, более четкое понимание часто приходит после изучения дополнительных данных и бесед с несколькими источниками.

Рекомендация аудитору 9: документируйте по ходу дела. Во время аудита важно вести заметки. Если откладывать это на потом, детали становятся расплывчатыми и неполными. Мне помогает вести черновые заметки по мере работы, а потом каждый вечер их приводить в порядок. Если отложить это больше чем на один-два вечера, вспоминать детали становится заметно сложнее.

Рекомендация аудитору 10: не теряйте доверие. При оценке данных вы увидите множество потенциальных проблем: какие-то из них могут оказаться серьезными стоп-факторами, а какие-то будут всего лишь мелкими ухабами. Обычно нужно время, чтобы откалибровать значимость замечаний. Если начать цепляться к мелочам, можно потерять внимание заявителя/разработчика в тот момент, когда вы обнаружите действительно серьезные проблемы. Когда я оцениваю относительно новый для себя проект, я обычно как минимум день или два держу свои мысли при себе, пока лучше не разберусь в данных. Иногда то, что сначала кажется серьезным, меркнет на фоне того, что открывается позже.

Рекомендация аудитору 11: учитывайте человеческий фактор. Одной из трудностей аудита являются, скажем так, *интересные личности*, которые встречаются по дороге. Аудиты сами по себе не особенно приятны ни для кого. Иногда стресс ситуации вытаскивает из людей худшее. Кто-то ведет себя конфронтационно. Кто-то вообще избегает сообщать скольконибудь полезную информацию. Кто-то нервный и напряженный. Другие дают обещания и не выполняют их. Это еще одна причина, почему полезно иметь напарника. Иногда, если у вас не получается найти подход к человеку, может получиться у напарника.

История о характерах #1: неприятные телеконы

Недавно я проводила дистанционный предварительный аудит этапа взаимодействия 1. Изначально это должен был быть аудит этапа взаимодействия 1, но планы были подготовлены настолько плохо, что я решила понизить его до предварительного этапа взаимодействия, чтобы дать команде возможность собраться и привести все в порядок до проведения *полноценного этапа взаимодействия*. Несмотря на мою доброжелательность, команда вела себя конфронтационно. Мне заявили, что это для них “не первый раз на стадионе” и что у них уже было несколько одобрений Федерального управления гражданской авиации США “за плечами”. Они вели себя так, будто я никогда не видела план программного обеспечения и не читала DO-178B. Я была одновременно ошарашена и слегка раздражена. Однако настойчивость дала результат. После двух раздражающих телеконференций, которые я называла

nasty-cons, мне удалось завоевать внимание команды, и мы смогли сосредоточиться на технических вопросах.

История о характерах #2: я боялась за свою жизнь

Несколько лет назад я проводила выездной аудит в одиночку. Результаты аудита были ужасными, и во всем мире не нашлось бы столько меда, чтобы сделать заключительный брифинг слаще. Я прямо сказала о выявленных проблемах. Я даже оформила замечание к SQA из-за практически полного отсутствия их активности (за пятилетний проект у них нашлось всего три записи SQA). После этого инженер по качеству программного обеспечения, которого я назову *Mister SQA*, спросил, может ли он поговорить со мной отдельно. Я согласилась, и он повел меня к себе в кабинет. К тому моменту уже было темно, а его кабинет находился на дальнем конце площадки. Мы шли через пустой цех, и я начала нервничать. Когда мы наконец пришли в кабинет, подальше от всех остальных, он начал на меня кричать. “Как вы можете писать нам такое?” – потребовал он. “Я могу из-за этого потерять работу!” Воображение тут же разыгралось. Я представила, как он вытаскивает бейсбольную битку, бьет меня по голове, режет тело на мелкие куски и закапывает под калибровочной машиной. Я немедленно начала искать кратчайший путь к отступлению. Я сказала несколько завершающих фраз и сообщила, что мне нужно в аэропорт на рейс. Я оставила *Mister SQA* с паром из ушей и была рада, что вообще ушла оттуда живой. Еще одна причина иметь свидетеля.

Рекомендация аудитору 12: используйте цели как мерило. На протяжении всего аудита важно использовать в качестве критериев оценки именно цели стандарта, либо DO-178В, либо DO-178С, в зависимости от того, что принято в качестве средства соответствия, а также любые применяемые дополнения. Вопросы из пособия и ваш опыт полезны, но оцениваете вы именно выполнение целей.

Рекомендация аудитору 13: сохраняйте фокус и доводите дело до конца. В большинстве аудитов есть хотя бы несколько отвлекающих факторов: данные могут быть неясными, люди – странными, а события – развиваться не по плану. В таких ситуациях важно сохранять сосредоточенность и докапываться до сути проблемы или проблем. Возможно, вам потребуется запросить дополнительное время, чтобы посмотреть данные в одиночку, или попросить кого-то еще провести вас по этим данным.

Рекомендация аудитору 14: сообщайте о возможных проблемах по ходу аудита. Когда вы уже поняли проект, процессы компании и заслужили уважение команды, полезно упоминать проблемы по мере их обнаружения. Это дает заявителю/разработчику шанс продумать стратегию исправления проблемы и, возможно, обсудить ее с вами еще до окончания аудита. Раньше я ждала заключительного доклада и разом вываливала все проблемы на голову заявителю/разработчику, но довольно быстро поняла, что подход “быстро доложил и убе-

жал” не работает. Есть разные способы сообщать команде о выявленных проблемах. Один вариант – проводить короткую встречу в конце дня, где вы делитесь *предварительными* замечаниями. Лучше отдельно подчеркнуть, что они именно *предварительные*, поскольку аудит еще продолжается. Другой вариант – устно упоминать проблемы сразу по мере их обнаружения, чтобы заявитель/разработчик мог вести собственный список. Важно делиться этой информацией с нужными людьми на протяжении всего аудита, чтобы заключительный брифинг и отчет по этапу взаимодействия стал лишь сводкой того, что уже обсуждалось.

Рекомендация аудитору 15: при необходимости переносите аудит или понижайте его статус. Иногда, сколько ни планируй и ни готовься, аудит начинается, и становится ясно, что данные просто не готовы, проще говоря, соответствия нет. В такой ситуации возможно несколько вариантов, например:

- Можно продолжить аудит и оформить несоответствия и замечания, которые заявитель-/разработчик должен будет устранить. Обычно позже аудит придется проводить заново.
- Можно понизить аудит до предварительного этапа взаимодействия или до неформальной оценки. Можно также остановить аудит и вернуться позже.

Существуют и другие варианты. Решение будет зависеть от нескольких факторов, включая личные предпочтения.

Рекомендация аудитору 16: быстро выдавайте отчет. После завершения аудита этапа взаимодействия важно как можно скорее предоставить отчет, чтобы заявитель/разработчик мог немедленно начать действовать. Обычно я включаю предварительный список несоответствий, замечаний и действий в заключительный брифинг, а официальный отчет отправляю в течение одной недели. В списке несоответствий я считаю полезным различать системные и единичные несоответствия. Системные несоответствия выявляются в нескольких местах и, следовательно, требуют действий в масштабе всего проекта. Единичные несоответствия тоже должны быть исправлены, но они не являются массовыми; устранить нужно только конкретный экземпляр.

Обычно я документирую несоответствия, замечания, действия и комментарии в табличном формате. Таблицу обычно оформляют в альбомной ориентации, чтобы у заявителя/разработчика было достаточно места для ответа. Ниже поясняется назначение каждого столбца:

- *Issue #* – этот столбец задает номер записи в таблице. Как и в случае с требованиями, если номер уже присвоен, лучше его не перенумеровывать, чтобы на него можно было ссылаться в других местах отчета. Например, если замечание удалено, номер сохраняется, а рядом дается пояснение, почему запись была удалена.
- *FOCA* – этот столбец классифицирует запись как несоответствие (F), замечание (O),

комментарий (C) или действие (A). Несоответствия и действия требуют реакции со стороны заявителя или разработчика. Замечание требует ответа, но не обязательно требует действий. Комментарий обычно является заметкой, которая может быть полезна при оценке соответствия. Комментарии не требуют ни ответа, ни действий со стороны заявителя или разработчика. Когда для определения классификации нужен отклик проекта, можно использовать обозначение “F?” или “?”. После получения ответа проекта классификацию можно уточнить.

- *Objective* – в этом столбце указывается цель или цели DO-178C, либо DO-178B, связанные с замечанием. В большинстве случаев достаточно номера таблицы приложения A DO-178C или DO-178B. Если вопрос является спорным, может понадобиться конкретный номер цели DO-178C или DO-178B и ссылка на раздел.
- *Systemic?* – значение ” Yes ” в этом столбце обозначает системную проблему.
- *Data* – в этом столбце указывается документ или элемент данных, к которому относится замечание. Если вопрос общий и не относится к конкретному документу, можно использовать слово ” General “. Где-то в отчете также должна быть указана версия данных.
- *Issue Description* – этот столбец содержит описание замечания. Оно должно быть конкретным. Указывайте раздел, номер требования, строку кода, фрагмент и так далее, и четко объясняйте, какая именно проблема выявлена.
- *Applicant/Developer Response* – этот столбец используется заявителем/разработчиком для ответа на замечание. Полезно датировать ответы и ставить инициалы, потому что до закрытия замечания иногда проходит несколько обновлений.
- *Evaluation* – этот столбец суммирует оценку замечания со стороны руководителя команды аудита этапа взаимодействия. Рекомендуется датировать комментарий оценки и ставить инициалы. Если ответ заявителя или разработчика требует от него дополнительных действий, ожидаемое действие поясняется. Если ответ приемлем, это нужно указать и закрыть замечание. Если ответ приемлем, но требует действий в будущем, возможно на следующем этапе взаимодействия, это тоже должно быть отмечено. Я обычно выделяю ответы, требующие будущих действий, синим цветом, чтобы не забыть вернуться к ним на следующем этапе взаимодействия.
- *Status* – столбец статуса также обновляется руководителем команды аудита этапа взаимодействия. Типичные значения: *Open, In-Work, Response Acceptable, Closed*. Цель состоит в том, чтобы все записи перешли в состояние *Closed*.

В таблице 12.3 приведена сводка разделов, которые я обычно включаю в отчет по этапу

взаимодействия.

Таблица 12.3 Пример структуры отчета по этапу взаимодействия

Раздел отчета	Содержание
1.0 Введение	Общий вводный раздел отчета по этапу взаимодействия.
1.1 Назначение	Краткое резюме проекта и назначение аудита этапа взаимодействия и отчета.
1.2 Обзор отчета	Общий обзор отчета.
1.3 Дата или даты аудита	Указывает, когда проводился аудит этапа взаимодействия.
1.4 Участники	Перечисляет как команду аудита, так и участников со стороны заявителя или разработчика.
1.5 Изученные данные	Перечисляет проверенные данные вместе с идентификацией конфигурации, то есть номерами документов и версиями.
2.0 Сводка несоответствий, замечаний, действий и комментариев	Включает описанную выше таблицу FOCA.
3.0 Вопросы пособия	Включает ответ на каждый вопрос пособия для конкретного аудита этапа взаимодействия. Обычно представляется в виде таблицы с использованием таблиц пособия и добавлением столбца оценки. Некоторые вопросы пособия могут оцениваться на следующем этапе взаимодействия; если это так, это следует отметить.
4.0 Оценка соответствия целям	Включает оценку соответствия каждой цели DO-178C или DO-178B.
5.0 Поддерживающая информация	Включает детали, которые могут использоваться в поддержку предыдущих разделов, например заметки с наблюдения за испытаниями или подробности по рассмотренным требованиям.

12.4.5 Общие рекомендации для аудируемой стороны (заявителя/разработчика)

Если вы являетесь аудируемой стороной, то есть заявителем или разработчиком программного обеспечения, которого проверяют, вот несколько рекомендаций.

Рекомендация аудируемой стороне 1: назначьте единое контактное лицо для взаимодействия с руководителем команды аудита этапа взаимодействия. Обычно это руководитель программного проекта. Если аудит этапа взаимодействия будет проводить сертифицирующий орган, контактным лицом может быть назначенный представитель проекта, например назначенный инженерный представитель. Контактное лицо будет координировать взаимодействие между руководителем команды аудита этапа взаимодействия и командой проекта.

Рекомендация аудируемой стороне 2: согласуйте с руководителем команды аудита этапа взаимодействия план и повестку. Назначенное контактное лицо координирует с руководителем команды аудита этапа взаимодействия логику, повестку и ожидания, чтобы они были хорошо понятны. Большую часть координации можно выполнить по электронной почте, однако иногда предпочтительнее телефонный звонок или телеконференция.

Рекомендация аудируемой стороне 3: проведите самоаудит и честно сообщайте о любых известных проблемах и о том, как они устраняются. До формального аудита этапа взаимодействия настоятельно рекомендуется провести пробный прогон, то есть предварительный этап взаимодействия. Если формальный аудит этапа взаимодействия будет проводить назначенный представитель, такой пробный прогон могут выполнить гарантия качества ПО и участник проектной команды. Если формальный аудит этапа взаимодействия будет проводить сертифицирующий орган, предварительный аудит, скорее всего, проведут назначен-

ный представитель или специалисты по взаимодействию с сертифицирующим органом.

Рекомендация аудируемой стороне 4: будьте готовы. Когда команда аудита этапа взаимодействия прибывает, данные и люди должны быть готовы. Аудиторам очень неприятно ждать, пока аудируемая сторона соберет людей и данные. Первое впечатление в аудите крайне важно и почти никогда не стирается. Вы точно не хотите, чтобы первым, что отметят аудиторы, стала *неподготовленность*. Если вы не уверены, какие данные ожидаются, проясните это заранее. Не все участники программной команды должны постоянно находиться в комнате на протяжении всего аудита, но они должны быть доступны в течение всего мероприятия. Если ключевой разработчик, верификатор или ведущий инженер будут отсутствовать во время аудита, заранее сообщите об этом руководителю команды аудита. Команда аудиторов может предпочесть перенести аудит. Я иногда шучу, насколько *удивительно случайно* совпадает мой приезд на аудит с тем, что у ключевых сотрудников внезапно находятся стоматолог на весь день, отпуск или похороны бабушки, причем уже в третий раз. Разумеется, конфликты расписаний бывают, но сделайте все возможное, чтобы ключевые люди были на месте. Обычно SQA и руководители команд участвуют во всем аудите.

Рекомендация аудируемой стороне 5: представляйте проект точно и кратко. В обычном аудите у заявителя/разработчика есть полдня, чтобы рассказать историю проекта. Используйте это время с умом, потому что именно оно задает тон остальному аудиту. Дайте точный и краткий обзор вашей системы, архитектуры программного обеспечения, используемых процессов и известных проблем. Если это этап взаимодействия 3, объясните подход к тестам. Кратко проведите аудиторов по требованиям, проекту, коду и данным тестирования, чтобы помочь команде аудита этапа взаимодействия понять подход проекта и структуру данных. Дайте такой фон и такой обзор, который позволит аудиторам эффективно выполнять свою работу. Будьте как можно более краткими, но при этом полными. Аудиторы могут раздражаться, если им покажется, что кто-то тратит их время впустую. Одна компания, которую я аудировала, попыталась растянуть презентацию на все три дня, хотя в повестке совершенно ясно было указано, что завершить ее нужно к полудню первого дня. Они выглядели *удивленными*, когда я сказала, что после обеда хочу уже смотреть данные.

Рекомендация аудируемой стороне 6: будьте вежливы, готовы к сотрудничеству и настроены позитивно. Аудиты никому не доставляют удовольствия, но их вполне можно пройти профессионально.

Команды заявителя или разработчика, которые ведут себя вежливо, сотрудничают и сохраняют позитивный настрой, почти всегда получают лучший отчет по этапу взаимодействия. Отношение *имеет* значение. Неконструктивное поведение вызывает подозрения и усиливает контроль, тогда как готовность к сотрудничеству и конструктивный настрой формируют

доверие.

Рекомендация аудируемой стороне 7: воспринимайте этот опыт как возможность улучшиться. Как и в любой области, встречаются отдельные люди с тяжелым характером, которые проводят такие аудиты. Однако большинство аудиторов обладают огромным опытом. Они видели множество проектов и могут быть ценнейшим источником знаний. Лучше смотреть на аудит этапа взаимодействия как на возможность научиться чему-то полезному, а не как на комнату пыток. Лучшие компании, с которыми мне доводилось работать, всегда стремились понять, как им стать лучше.

Рекомендация аудируемой стороне 8: убедитесь, что команда полностью понимает все несоответствия, включая их основу в DO-178C или DO-178B. Важно понимать несоответствия и действия, задокументированные командой аудита этапа взаимодействия. Иногда формально записанное несоответствие на деле оказывается просто чьим-то мнением. Если вы подозреваете, что это так, тактично уточните, какая именно цель DO-178C или DO-178B не выполняется. Также обязательно поймите, какие несоответствия являются системными, то есть относятся ко всему проекту, а какие являются единичными, поскольку это существенно влияет на ответы. Если есть сомнения, просите разъяснения.

Рекомендация аудируемой стороне 9: относитесь к несоответствиям серьезно и контролируйте их устранение. По определению, несоответствие должно быть устранено до сертификации. Следовательно, действия обязательны.

Много лет назад, когда я была сравнительно новым инженером Федерального управления гражданской авиации США, я проводила аудит, эквивалентный этапу взаимодействия 2, это было еще до появления пособия, поэтому сам термин SOI тогда еще не использовался, для системы управления полетом в компании, которая выполняла свой первый проект по DO-178B. До этого у них было несколько военных проектов, но взаимодействие с управлением и сам стандарт DO-178B они только осваивали. Я провела аудит и выдала отчет с многочисленными системными несоответствиями. У проекта был назначенный инженерный представитель, поэтому дальнейшую работу я оставила на него. Через год я вернулась для эквивалента этапа взаимодействия 3. Я была поражена, когда узнала, что за это время они так и не устранили ни одного замечания из аудита уровня этапа взаимодействия 2. Для всех это обернулось катастрофой и повлияло на общий график сертификации. Из этой ситуации все вынесли уроки, включая меня. Теперь я контролирую дальнейшие действия, даже если формально это и не входит в мои обязанности, чтобы убедиться, что работа действительно ведется.

Рекомендация аудируемой стороне 10: документируйте, как команда будет закрывать все несоответствия, действия и замечания. Как можно раньше начните готовить ответ на заме-

чания этапа взаимодействия. Обычно по итогам аудита этапа взаимодействия формируется отчет со списком несоответствий, действий и замечаний. Часто этот список уже оформлен в виде таблицы, и тогда можно просто добавить столбец с ответом проекта. Если список не представлен в табличной форме, я рекомендую перевести его в такую форму. Некоторые компании используют для этого электронную таблицу, поскольку в ней удобно сортировать и фильтровать записи. Обязательно дайте ответ на все несоответствия, действия и замечания. Замечания можно не устранять, но ответ на них все равно нужен. Если у вас есть вопросы, запросите разъяснение у руководителя команды аудита этапа взаимодействия или у своего назначенного представителя. Когда будет подготовлен ответ на все несоответствия, действия и замечания, передайте его руководителю команды аудита этапа взаимодействия для получения обратной связи. Очень часто для согласования всех вопросов требуется несколько итераций. В некоторых ситуациях может оказаться проще назначить телеконференцию или встречу и обсудить ответы там, чем бесконечно обмениваться письмами.

12.4.6 Особенности аудитов этапов взаимодействия

В этом разделе более подробно рассматриваются все четыре аудита этапа взаимодействия. Обсуждаются следующие темы:

- типичные входные критерии для аудита этапа взаимодействия, то есть что должно быть сделано до того, как такой аудит можно будет проводить;
- что команда аудита этапа взаимодействия обычно делает во время аудита, и как к нему подготовиться.

12.4.6.1 Входные критерии, ожидания и рекомендации по подготовке для этапа взаимодействия 1

12.4.6.1.1 Этап взаимодействия 1: когда он проводится

Аудит этапа взаимодействия 1 проводится после того, как планы и стандарты были рассмотрены и утверждены заявителем/разработчиком в качестве базовой версии. Если аудит этапа взаимодействия выполняет сертифицирующий орган, он обычно захочет рассматривать выпущенные данные. Если аудит этапа взаимодействия выполняет назначенный представитель, он может оценивать еще не выпущенные данные, но они все равно должны быть утверждены в качестве базовой версии. Ожидания в отношении выпуска данных следует заранее прояснить с руководителем аудита этапа взаимодействия.

12.4.6.1.2 Этап взаимодействия 1: чего ожидать

Аудит этапа взаимодействия 1 часто проводится дистанционно. В этом случае руководитель команды аудита этапа взаимодействия запросит предоставление планов и стандартов. Обычно команде аудита этапа взаимодействия требуется не менее месяца, чтобы изучить эти данные. Затем сводка замечаний представляется в письменном виде, как правило, в

отчете по аудиту этапа взаимодействия. Также могут проводиться телеконференция или встреча для обсуждения выявленных замечаний. Ниже приведено то, что команда аудита этапа взаимодействия обычно делает:

- убеждается, что планы и стандарты находятся под управлением конфигурацией до начала их оценки. Очень неприятно потратить 40 часов на рассмотрение набора данных, а затем обнаружить, что все изменилось без отслеживания изменений, создав аудиторам дополнительную работу;
- использует разделы 11.1-11.8 DO-178C, чтобы убедиться, что в планах и стандартах присутствует все ожидаемое содержимое. Данные могут быть оформлены иначе, чем предлагает DO-178C, например стандарты могут быть включены в План разработки ПО (SDP), а План разработки ПО (SDP) и План верификации ПО (SVP) могут быть объединены, однако базовое содержимое, указанное в разделах 11.1-11.8 DO-178C, все равно должно присутствовать в планах и стандартах;
- использует цели приложения A DO-178C, чтобы убедиться, что в планах охвачены все применимые цели. Аудиторы обычно проверяют, что план сертификации ПО охватывает все применимые цели. План сертификации ПО может не вдаваться в детали, но должен хотя бы в общем виде показывать, как будет обеспечиваться выполнение каждой цели. Аудиторы также убеждаются, что План разработки ПО (SDP), План верификации ПО (SVP), План управления конфигурацией ПО (SCMP) и План гарантии качества ПО (SQAP) содержат детали того, как эти цели будут выполняться;
- рассматривает детали *дополнительных соображений*, например квалификации инструментов. Аудиторы убеждаются, что такие дополнительные соображения объяснены достаточно полно, а описанный подход является приемлемым;
- убеждается, что планы согласованы. Аудиторы проверяют как внутреннюю согласованность каждого отдельного плана, так и внешнюю согласованность, то есть согласованность всех планов между собой;
- убеждается, что в планах отражены проблемные записки или их эквиваленты. В некоторых случаях, например в проектах по техническому стандартному приказу, номера проблемных записок могут прямо не указываться, но все равно должны быть признаки того, что соответствующие вопросы рассматриваются. Например, даже если проблемная записка по модельно-ориентированной разработке не упомянута в плане сертификации ПО, в нем все равно может быть раздел, объясняющий, как в проекте выполняется модельно-ориентированная разработка. Для программного обеспечения, привязанного к конкретному воздушному судну или двигателю, проблемные записки по ПО либо их эквиваленты и ответ проекта на них обычно суммируются в плане сертификации ПО;

- проверяет стандарты, чтобы убедиться, что они существуют, пригодны для использования, реально используются и подходят для данного проекта;
- оценивает планы квалификации инструментов, если они оформлены отдельно от плана сертификации ПО, с использованием DO-330. Если инструмент разрабатывался до официального признания DO-178C и DO-330, применяются критерии DO-178B и FAA Order 8110.49 либо EASA CM-SWCEH-002;
- задает вопросы о том, как команда использует планы и стандарты. Аудиторы могут захотеть понять, каким образом проект обеспечивает следование этим планам. Во многих компаниях обязательны обучение и ознакомительное чтение. Во время аудита могут изучаться записи об обучении и ознакомлении.

12.4.6.1.3 Этап взаимодействия 1: как подготовиться

Ниже приведены некоторые рекомендации по подготовке к этапу взаимодействия 1:

- завершите все планы и стандарты, используя рекомендации разделов 11.1-11.8 DO-178C;
- завершите планы квалификации инструментов, если они нужны, используя рекомендации DO-330. Проведите коллегиальное рассмотрение планов и стандартов, включая совместное рассмотрение всего комплекта;
- обеспечьте согласованность между планами. Обычно это оценивается в ходе коллегиального рассмотрения планирования;
- выполните сопоставление с целями DO-178C. Как отмечалось в главе 5, полезно предоставить сопоставление между целями DO-178C, Планом сертификации ПО (PSAC) и другими планами. Если используются какие-либо дополнения к DO-178C, планы должны быть сопоставлены и с этими целями;
- устраните все проблемы планирования до аудита этапа взаимодействия. Любые замечания, выявленные во время коллегиального рассмотрения, должны быть закрыты до формального аудита этапа взаимодействия. Переведите планы под управление конфигурацией;
- убедитесь, что стандарты существуют, были рассмотрены по критериям DO-178C, применимы к проекту и используются командой разработки;
- убедитесь, что все члены команды знают планы, процедуры и стандарты и действительно им следуют;
- учитывайте вопросы пособия при разработке и рассмотрении планов;

- подготовьте ответы на все вопросы пособия до формального аудита этапа взаимодействия.

12.4.6.2 Входные критерии, ожидания и рекомендации по подготовке для этапа взаимодействия 2

12.4.6.2.1 Этап взаимодействия 2: когда он проводится

Обычно этап взаимодействия 2 проводится после того, как разработано и прошло рассмотрение не менее 50% кода. Иногда предварительный этап взаимодействия 2, то есть неформальное мероприятие для снижения риска, может проводиться раньше, однако формальный этап взаимодействия 2 обычно требует более зрелого продукта.

12.4.6.2.2 Этап взаимодействия 2: чего ожидать

Этап взаимодействия 2 обычно проводится на месте. В зависимости от размера и характера проекта он может фактически выполняться в несколько этапов. Например, я была консультирующим назначенным инженерным представителем и руководителем команды аудита этапа взаимодействия на одном проекте, разделенном на 47 функций, каждая размером от 500 до 5000 строк кода. Поскольку функции разрабатывали разные команды в разных местах, а проект считался высокорисковым, Федеральное управление гражданской авиации США потребовало провести аудит каждой из 47 функций. Поэтому этап взаимодействия 2 разделили на пять частей, чтобы охватить все функции, а некоторые проблемные функции проверяли несколько раз. На проектах, где первый аудит этапа взаимодействия 2 выявляет многочисленные системные несоответствия, аудиты этапа взаимодействия 2 продолжают-ся до тех пор, пока проблемы не будут устранены. Я сравниваю это с выпечкой пирога: вы снова и снова втыкаете зубочистку, то есть проводите рассмотрения этапа взаимодействия 2, пока пирог окончательно не пропечется и зубочистка не начнет выходить чистой, то есть пока больше не будут выявляться системные проблемы.

Ниже приведена сводка того, что команда аудита этапа взаимодействия обычно делает во время аудита этапа взаимодействия 2:

1. Закрывает отчет по аудиту этапа взаимодействия 1. Все замечания, не закрытые после аудита этапа взаимодействия 1, обычно обсуждаются и закрываются в начале аудита этапа взаимодействия 2. Предполагается, что большинство замечаний этапа взаимодействия 1 будет устранено до аудита этапа взаимодействия 2, поскольку закрытие аудита этапа взаимодействия 1 считается предварительным условием для подачи плана сертификации ПО в сертифицирующий орган. Однако возможны проекты, где аудит этапа взаимодействия 1 проводится поздно и фактически переходит прямо в аудит этапа взаимодействия 2.
2. Просит заявителя/разработчика пройти одну нисходящую цепочку требований: от си-

системных требований к программным требованиям, затем к проекту ПО и исходному коду, а также одну восходящую цепочку: от исходного кода к проекту ПО, затем к программным требованиям и системным требованиям. Как отмечалось ранее, цель этого упражнения – познакомить аудитора с данными заявителя/разработчика и механизмом трассируемости.

3. Выбирает несколько цепочек требований и выполняет нисходящие и восходящие проверки согласованности. Аудиторы этапа взаимодействия обычно выбирают разнообразные цепочки, учитывая следующее:
 - a. *Функциональность*: выбираются цепочки из разных функциональных областей.
 - b. *Команда разработки*: выбираются выборки данных из разных команд разработки, чтобы убедиться, что каждая команда следует определенным процессам и стандартам.
 - c. *Сложность*: выбираются как простые цепочки, так и сложные. Некоторые команды отлично справляются со сложными функциями, но теряют дисциплину на простых, или наоборот.
 - d. *Известные проблемы*: если имеются известные проблемные области, выявленные лабораторными испытаниями, испытаниями на воздушном судне или сообщениями о проблемах, аудиторы могут выбрать данные из этой функциональной области, чтобы определить, нет ли там системных проблем.
 - e. *Функции, связанные с безопасностью*: часто выбираются требования, наиболее важные для безопасности, чтобы убедиться, что они реализуются надлежащим образом.
4. Оценивает полноту и точность трассируемости. Это делается в процессе прохождения по цепочкам требований.
5. Ищет несогласованности между требованиями, проектом и кодом. Аудиторы также оценивают, полностью ли требования низкого уровня и/или код реализуют требования, к которым они восходяще трассируются.
6. Изучает данные квалификации инструментов для любых инструментов, используемых в процессе разработки и требующих квалификации. Кроме того, может выполняться оценка всех инструментов, чтобы подтвердить корректность решения о необходимости квалификации, то есть убедиться, что все инструменты, которым нужна квалификация, действительно квалифицируются.
7. Оценивает соблюдение планов и стандартов.

8. Изучает записи рассмотрений и заполненные контрольные перечни, чтобы убедиться, что рассмотрения были тщательными и использовались подходящие контрольные перечни.
9. Рассматривает сообщения о проблемах и/или запросы на изменения, а также сам процесс внесения изменений.
10. Убеждается, что используется указанная среда разработки, зафиксированная в Указателе конфигурации среды ЖЦ ПО или указателе конфигурации ПО, включая настройки и опции компилятора.
11. Оценивает процессы управления конфигурацией ПО.
12. Наблюдает за процессом сборки, чтобы убедиться, что документированные процедуры воспроизводимы.
13. Изучает записи по гарантии качества ПО и беседует с персоналом гарантии качества ПО.
14. Если некоторые тестовые примеры уже существуют, аудиторы могут выбрать их выборочно, чтобы убедиться, что усилия по тестированию движутся в правильном направлении. Во время аудита этапа взаимодействия 2 это неформальная оценка, но она полезна для ранней обратной связи и снижения риска перед этапом взаимодействия 3.

12.4.6.2.3 Этап взаимодействия 2: как подготовиться

Ниже приведены некоторые рекомендации по подготовке к аудиту этапа взаимодействия 2:

Назначьте контактное лицо для координации с руководителем команды аудита этапа взаимодействия.

- убедитесь, что замечания предыдущего аудита этапа взаимодействия устранены. Предоставьте ответы руководителю команды аудита этапа взаимодействия до проведения аудита этапа взаимодействия 2;
- убедитесь, что все данные доступны. В зависимости от размера команды аудита для этого может понадобиться несколько рабочих станций. Некоторые аудиторы могут запросить бумажные копии части данных, поэтому при необходимости будьте готовы их распечатать;
- проведите пробный аудит этапа взаимодействия 2 с использованием вопросов пособия Software Job Aid. ЗадOCUMENTИРУЙТЕ ответы на вопросы пособия для команды аудита. Определите все известные проблемы, включая выявленные во время пробного прогона, и план их устранения;

- выполните несколько предварительных проходов по цепочкам требований, чтобы убедиться, что данные готовы. Это может быть частью пробного аудита этапа взаимодействия 2. Позже эти предварительные цепочки могут использоваться как примеры во время презентации для команды аудита этапа взаимодействия;
- убедитесь, что команда разработки следует планам и стандартам. Выявите все отклонения или недостатки, а также запланированные способы их устранения;
- подготовьте презентацию по системе, используемым процессам, текущему состоянию и всем известным проблемам, вместе с уже выполняемыми корректирующими действиями;
- убедитесь, что записи рассмотрений полны и доступны;
- подготовьте данные квалификации инструментов, если квалифицируются какие-либо инструменты, используемые в процессах разработки, например генератор кода или средство проверки соответствия стандартам кодирования;
- убедитесь, что данные трассировки точны и доступны;
- согласуйте повестку аудита и убедитесь, что все члены команды понимают, чего ожидать. Встреча со всей командой разработки очень полезна. Если команда впервые проходит аудит, полезно заранее объяснить, как на него отвечать. Побуждайте людей отвечать честно и точно и отвечать только на прямые вопросы;
- обеспечьте доступность подходящих разработчиков программного обеспечения во время аудита. Подключите службу гарантии качества ПО (SQA) к подготовке к этапу взаимодействия 2 и к самому мероприятию этапа взаимодействия 2, поскольку именно она может отвечать за то, чтобы исправления действительно вносились.

12.4.6.3 Входные критерии, ожидания и рекомендации по подготовке для этапа взаимодействия 3

12.4.6.3.1 Этап взаимодействия 3: когда он проводится

Аудит этапа взаимодействия 3 обычно проводится после того, как разработано и прошло рассмотрение не менее 50% тестовых примеров и процедур. Кроме того, во время аудита этапа взаимодействия 3 нужны примеры данных или четко определенный подход для следующих видов анализа, в зависимости от уровня ПО: структурное покрытие, межкомпонентная связность по данным и по управлению, наихудшее время выполнения, использование стека, запас по памяти, прерывания, трассируемость от исходного кода к объектному коду и так далее.

12.4.6.3.2 Этап взаимодействия 3: чего ожидать

Как и аудит этапа взаимодействия 2, аудит этапа взаимодействия 3 обычно проводится на

месте и может проходить в несколько этапов в зависимости от размера проекта и возникших трудностей. Ниже приведена сводка того, что команда аудита этапа взаимодействия 3 обычно делает на месте:

1. Рассматривает открытые вопросы из предыдущих аудитов этапов взаимодействия и либо закрывает их, либо определяет, какие дополнительные действия нужны.
2. Изучает дополнительные данные, разработанные после аудита этапа взаимодействия 2, например новые требования, проект и код, чтобы убедиться, что использовались те же процессы или что проблемы процесса устранялись последовательно.
3. Выбирает некоторые требования высокого уровня и изучает соответствующие им тестовые примеры и процедуры, чтобы убедиться, что у них есть следующие характеристики:
 - a. *Трассируемость* – убедиться, что существует двусторонняя трассируемость между требованиями и тестовыми примерами, между тестовыми примерами и тестовыми процедурами, а также между тестовыми процедурами и результатами тестирования, и что она точна.
 - b. *Полнота* – убедиться, что тестируется все требование целиком.
 - c. *Робастность* – убедиться, что требование, где это уместно, проверено на робастность.
 - d. *Адекватность* – убедиться, что тест действительно корректно проверяет требование, является эффективным, содержит критерии прохождения и непрохождения и так далее.
 - e. *Воспроизводимость* – убедиться, что тестовые процедуры ясны и могут быть выполнены тем, кто их не писал, а также что при каждом запуске теста будут получаться одинаковые результаты.
 - f. *Успешное прохождение* – убедиться, что тесты дают ожидаемые результаты. На этой стадии обычно рассматриваются только неформальные результаты, поскольку формальные результаты анализируются на этапе взаимодействия 4.
4. Выбирает некоторые требования низкого уровня и изучает соответствующие им тестовые данные, чтобы убедиться, что они имеют те же характеристики, что и тесты по требованиям высокого уровня: трассируемость, полноту, робастность, адекватность, воспроизводимость и успешное прохождение.
5. Изучает записи рассмотрений тестовых примеров и процедур, чтобы убедиться, что рассмотрения были тщательными и соответствующие контрольные перечни были за-

полнены.

6. Изучает данные структурного покрытия, чтобы убедиться, что покрытие правильно измеряется и анализируется. Анализ структурного покрытия, скорее всего, еще будет в работе, но часть данных уже будет изучаться, чтобы убедиться, что выбранный подход позволяет выполнить цели DO-178C.
7. Изучает имеющиеся данные по интеграционным видам анализа, таким как межкомпонентная связность по данным и по управлению, временной анализ и анализ памяти. Эти виды анализа могут еще выполняться, но общий подход уже должен быть определен, а часть данных должна быть подготовлена в черновом виде. Особое внимание будет уделяться воспроизводимости этих анализов.
8. Изучает, насколько корректно во время тестирования обеспечивается интеграция, требуемая DO-178C, то есть интеграция ПО с ПО и ПО с аппаратурой. Если во время тестирования используется много точек останова или если испытания проводятся по модулям, а не в интегрированном виде, подход к интеграции будет анализироваться особенно тщательно.
9. Убеждается, что проверены все требования. Если используется анализ или инспекция кода, этот подход будет рассмотрен, чтобы определить, достаточен ли он для доказательства того, что исполняемый объектный код удовлетворяет установленным требованиям.
10. Оценивает данные квалификации инструментов для любых инструментов, используемых в процессе верификации и требующих квалификации. Другие инструменты также могут изучаться, чтобы подтвердить, что квалификация для них действительно не нужна.
11. Оценивает сообщения о проблемах, чтобы убедиться, что они заполнены полностью, включая описание, анализ, исправление и верификацию изменений в данных разработки и тестовых данных.
12. Оценивает изменения, реализованные по результатам мероприятий по верификации.
13. Наблюдает за выполнением тестов, чтобы убедиться в воспроизводимости.
14. Определяет, находятся ли тестовые и верификационные данные под управлением конфигурации и управлением изменениями.
15. Изучает записи по управлению конфигурацией ПО для требований, тестовых примеров и процедур, которые были изменены или добавлены.
16. Изучает данные гарантии качества ПО (SQA), связанные с процессом тестирования и

интеграционными анализами, и беседует с персоналом гарантии качества ПО (SQA).

17. Оценивает корректность среды верификации, то есть убеждается, что она соответствует среде, указанной в Указателе конфигурации среды ЖЦ ПО или указателе конфигурации ПО.

12.4.6.3.3 Этап взаимодействия 3: как подготовиться

Ниже приведены некоторые рекомендации по подготовке к этапу взаимодействия 3:

- назначьте контактное лицо для координации с руководителем команды аудита этапа взаимодействия. Убедитесь, что вопросы из предыдущих этапов взаимодействия уже отработаны, и будьте готовы обсуждать их с аудиторами;
- будьте готовы представить обзор подхода к тестам, тестовых данных, текущего состояния, известных проблем, подхода к структурному покрытию и других видов анализа;
- убедитесь, что тестовые примеры и процедуры верификации ПО прошли рассмотрение. Если рассмотрены не все, четко обозначьте, какие именно уже рассмотрены, поскольку аудит этапа взаимодействия обычно сосредотачивается именно на них;
- подготовьте для рассмотрения тестовые примеры и процедуры верификации ПО, а также записи рассмотрений. Если помимо плана верификации ПО существует отдельный План тестов, предоставьте его аудиторам и кратко опишите его содержание;
- предоставьте результаты верификации. На данном этапе проекта это могут быть результаты пробных прогонов или даже более неформальных запусков. Аудиторы в первую очередь захотят увидеть, как вы собираетесь выполнять тесты и документировать результаты, чтобы убедиться, что они организованы, полны и воспроизводимы;
- будьте готовы запустить выбранные тесты в присутствии аудиторов. Выполняйте тесты по процедурам, а не по памяти;
- подготовьте двусторонние данные трассировки между требованиями и тестовыми примерами верификации, между тестовыми примерами верификации и процедурами верификации, а также между процедурами верификации и результатами верификации, если такие результаты уже существуют. Если результатов пока нет, покажите пример того, как будет выполняться трассируемость к результатам. Будьте готовы показать тестовые данные как для обычных проверок, так и для проверок робастности, причем и для требований высокого уровня, и для требований низкого уровня, кроме уровня D, где тестирование требований низкого уровня не требуется;
- подготовьте данные структурного покрытия, межкомпонентной связности по данным и по управлению, а также других видов анализа. Убедитесь, что технический эксперт

по каждому виду анализа доступен и готов объяснить выбранный подход;

- убедитесь, что тестировщики используют именно ту среду испытаний, которая указана в Указателе конфигурации среды ЖЦ ПО или указателе конфигурации ПО;
- подготовьте данные квалификации инструментов к рассмотрению, если инструменты квалифицируются;
- заранее изучите вопросы пособия Software Job Aid, выявите все известные проблемы и примите меры для их устранения. Подготовьте ответы на вопросы пособия для аудиторов.

12.4.6.4 Входные критерии, ожидаемое содержание и рекомендации по подготовке для этапа взаимодействия 4

12.4.6.4.1 Этап взаимодействия 4: когда он проводится

Этап взаимодействия 4 проводится после того, как устранены замечания предыдущих аудитов этапов взаимодействия, а результаты верификации, Указатель конфигурации ПО и Итоговый отчет разработки ПО рассмотрены и включены в базовую конфигурацию. Если аудит этапа взаимодействия выполняет сертифицирующий орган, обычно требуются выпущенные данные. Если аудит этапа взаимодействия выполняет назначенный представитель, он может оценивать предварительно выпущенные данные (но уже включенные в базовую конфигурацию) во время аудита, а выпущенные данные проверить до закрытия этого этапа.

12.4.6.4.2 Этап взаимодействия 4: чего ожидать

Этап взаимодействия 4 часто проводится дистанционно, но некоторые аудиторы предпочитают завершать его на месте, особенно если в ходе предыдущих этапов взаимодействия было много замечаний. Решение о том, будет ли аудит выездным или дистанционным, зависит от характера проекта и предпочтений аудитора или аудиторов. Ниже перечислено, что обычно делает команда аудита этапа взаимодействия 4:

1. Проверяет все выводы, действия и замечания из предыдущих этапов взаимодействия, чтобы убедиться, что они удовлетворительно закрыты, и закрывает отчет по этапу взаимодействия 3 (а также любые другие отчеты по этапам взаимодействия, которые раньше не были закрыты).
2. Изучает дополнительные данные, разработанные после последнего аудита этапа взаимодействия, чтобы убедиться, что применялись те же процессы или что проблемы процесса устранялись последовательно (например, новые тестовые примеры и процедуры, результаты тестирования, завершенные данные структурного покрытия и результаты различных анализов).
3. Изучает результаты верификации ПО.

4. Оценивает анализы и обоснование для любого кода, не охваченного анализом структурного покрытия.
5. Изучает окончательные версии Указателя конфигурации ПО и Указателя конфигурации среды ЖЦ ПО на предмет корректности и согласованности с выпущенными данными.
6. Убеждается, что все сообщения о проблемах должным образом закрыты или по ним оформлено решение.
7. Рассматривает Итоговый отчет разработки ПО и убеждается в следующем:
 - a. Задokumentированы все согласованные особые условия или договоренности (например, посторонний код или ограничения системы).
 - b. Задokumentированы отклонения от планов.
 - c. Для открытых или отложенных сообщений о проблемах выполнен анализ влияния на безопасность, характеристики, эксплуатацию или соответствие нормативным требованиям.
 - d. Итоговый отчет разработки ПО соответствует тому, что фактически происходило в ходе проекта.
8. Изучает Указатель конфигурации инструмента и Итоговый отчет разработки инструмента, если использовались какие-либо инструменты уровней TQL-1[*] – TQL-4 либо если инструмент TQL-5 имеет Указатель конфигурации инструмента и/или Итоговый отчет разработки инструмента, оформленные отдельно от Итогового отчета разработки ПО. Информацию о квалификации инструментов см. в главе 13.
9. Изучает записи по управлению конфигурацией ПО.
10. Изучает записи по гарантии качества ПО.
11. Изучает результаты рассмотрения конформности.

12.4.6.4.3 Этап взаимодействия 4: как подготовиться

Ниже приведены некоторые рекомендации по подготовке к этапу взаимодействия 4:

- убедитесь, что замечания предыдущих этапов взаимодействия устранены, и будьте готовы обсуждать их;
- завершите Указатель конфигурации ПО, Указатель конфигурации среды ЖЦ ПО, Отчет о результатах верификации ПО и Итоговый отчет разработки ПО; проведите их рассмотрение; устраните замечания, выявленные в ходе рассмотрений;
- убедитесь, что рассмотрение конформности ПО завершено и все выявленные проблемы устранены. Подготовьте записи рассмотрения конформности. Будьте готовы показать

любые данные, которые не оценивались на предыдущих этапах взаимодействия (например, окончательные результаты верификации, данные структурного покрытия, данные анализа межкомпонентной связности по данным и по управлению, анализ наихудшего времени выполнения, анализ использования стека, анализ компоновки, анализ отображения памяти, анализ загрузки и анализ от исходного кода к объектному коду). Заранее изучите вопросы пособия; устраните все выявленные проблемы; подготовьте ответы на вопросы пособия для аудитора. Обязательно учтите также те вопросы пособия, на которые не были даны ответы во время предыдущих аудитов этапов взаимодействия (например, вопросы этапа взаимодействия 3).

12.5 Зрелость программного обеспечения перед сертификационными летными испытаниями

Один из вопросов, который неизменно возникает в ходе сертификации, звучит так: «Насколько зрелым должно быть программное обеспечение, прежде чем его можно будет использовать для официальных летных испытаний?» ПО должно быть достаточно подтверждено до того, как можно будет успешно проводить испытания на уровне системы и воздушного судна. Существуют сертификационные нормы и руководящие материалы, которые по существу требуют, чтобы система находилась в состоянии, пригодном для сертификации, до официальных летных испытаний FAA.[*]

В идеале все мероприятия по подтверждению соответствия DO-178C должны быть завершены до сертификационных летных испытаний системы или систем, использующих данное программное обеспечение. Сертифицирующие органы явно предпочитают именно такой уровень зрелости. Однако, поскольку ПО часто оказывается одним из последних элементов, достигающих зрелости, идеальное состояние часто недостижимо. Поэтому сертифицирующие органы выработали критерии зрелости ПО. На момент написания этой книги критерии зрелости ПО еще не были окончательно закреплены и поэтому могут несколько различаться от проекта к проекту. Тем не менее существуют некоторые общие рекомендации, которые обычно используются как отправная точка для согласования с сертифицирующим органом. Для того чтобы убедиться, что ПО достаточно зрелое для сертификационных летных испытаний, обычно используются следующие критерии [5]:

1. Аудиты этапов взаимодействия 1-3 проведены, и все существенные замечания, то есть несоответствия, закрыты. Существенными считаются замечания рассмотрений, которые могут повлиять на безопасность, характеристики, эксплуатацию и/или вынесение решения о соответствии; следовательно, они должны быть устранены до летных испытаний с этим ПО.
2. До сертификационных летных испытаний данного программного обеспечения оно

должно находиться в состоянии, эквивалентном производственной версии (black label). Это означает, что имеется высокая уверенность в том, что ПО на летно-испытательном воздушном судне является тем самым ПО, которое фактически будет установлено на сертифицированные и серийные воздушные суда. Уверенность в том, что ПО эквивалентно производственной версии, обычно требует следующего:

- a. Все системные требования, распределенные на ПО, реализованы в ПО.
- b. Все тесты ПО, основанные на требованиях, включая тесты по требованиям высокого и низкого уровней, нормальные тесты и тесты на робастность, выполнены на целевой платформе.[*] Выполнение тестов ПО означает следующее: (1) тестовые примеры и процедуры прошли рассмотрение; (2) тестовая среда, тестовые примеры и процедуры, тестовые сценарии и тестируемое ПО находятся под управлением конфигурацией; (3) тесты выполнены, а результаты сохранены и находятся под управлением; (4) по всем отказам тестов заведены сообщения о проблемах.
- c. Все существенные сообщения о проблемах, влияющие на функциональность и безопасность, исправлены, проверены и включены в ПО, которое будет использоваться в летных испытаниях.
- d. Указатель конфигурации ПО сформирован, соответствует разделу 11.16 DO-178C и поставляется вместе с программным обеспечением, устанавливаемым для летных испытаний.

Любое программное обеспечение, не соответствующее этим критериям, обычно требует специального согласования с сертифицирующим органом до сертификационных летных испытаний. По этому вопросу обычно бывает довольно много согласований. Во многих случаях ПО признается достаточно зрелым, летные испытания выполняются, а затем ПО изменяется. В таком сценарии часть или даже все летные испытания могут потребоваться выполнить повторно по результатам анализа влияния изменений на уровне системы и ПО.

Ссылки

- 1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
- 2. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Change 1, September 2011).
- 3. European Aviation Safety Agency, *Software Aspects of Certification*, Certification Memorandum CM-SWCEH-002 (Issue 1, August 2011).
- 4. Federal Aviation Administration, *Conducting Software Reviews Prior to Certification*, Aircraft Certification Service (Rev. 1, January 2004).

5. W. Struck, Software maturity, presentation at 2009 Federal Aviation Administration National Software and Complex Electronic Hardware Conference (San Jose, CA, August 2009).

*Заявитель – это сторона, подающая заявку на сертификацию или авторизацию техническому стандартному приказу. Заявитель отвечает за демонстрацию соответствия нормативным требованиям. Когда DO-178C выбран в качестве метода определения соответствия, заявитель также отвечает за демонстрацию выполнения целей и руководящих указаний DO-178C.

*Квадратные скобки ([]) указывают на последнюю редакцию. Эти документы время от времени обновляются. Последнюю версию можно найти на сайте FAA по адресу www.faa.gov.

†Раздел 4 сертификационного меморандума EASA CM-SWCEH-002 содержит аналогичную информацию для проектов, сертифицируемых в Европе [3].

*После того как сертифицирующие органы официально признают DO-178C, DO-330 и дополнения, несколько типовых проблемных записок по программному обеспечению, вероятно, больше не понадобятся.

*XX может означать части 23, 25, 27 или 29 раздела 14 Свода федеральных нормативных актов США (Code of Federal Regulations, CFR), обычно сокращаемого как 14 CFR.

*Исходная версия пособия была выпущена в июне 1998 года. Rev 1 была выпущена в январе 2004 года. Ожидается, что пособие будет обновлено для согласования с DO-178C, DO-330 и дополнениями. Актуальную версию Software Review Job Aid см. на сайте FAA: www.faa.gov.

†Глава 4 сертификационного меморандума EASA CM-SWCEH-002 очень похожа на главу 2 приказа FAA Order 8110.49.

‡ *Auditee* – это организация, проходящая аудит.

*На момент написания этой книги пособие ссылается на цели DO-178B, а не DO-178C. Большая часть пособия по-прежнему применима к DO-178C. Ожидается, что Федеральное управление гражданской авиации США обновит пособие, чтобы привести его в соответствие с DO-178C.

*Некоторые личные и отчасти юмористические истории включены в выделенные рамкой вставки.

*Выездной аудит проводится на площадке разработчика или иногда на площадке заявителя, если заявитель и разработчик не совпадают. Настольный аудит выполняется дистанционно путем изучения данных. Во многих случаях этапы взаимодействия 1 и 4 могут выполняться дистанционно, тогда как этапы взаимодействия 2 и 3 проводятся на площадке.

*TQL – это уровень квалификации инструмента, назначаемый на этапе планирования.

*Нормативные требования включают части 21.33(b), 21.35(a) и XX.1301 раздела 14 Свода

федеральных нормативных актов США (Code of Federal Regulations, CFR), обычно обозначаемого как 14 CFR, где XX может означать 23, 25, 27 или 29. Дополнительные рекомендации также приведены в разделе 5-19, пункты d-f, приказа FAA Order 8110.4[].

*Формально выполненные испытания программного обеспечения предпочтительны; однако бывали ситуации, когда принимались результаты пробных прогонов, если такой прогон был полным, то есть охватывал все требования, и успешным, то есть без неприемлемых отказов.

Часть IV. Квалификация инструментов и дополнения к DO-178C (KT-178C)

В части IV рассматриваются четыре руководящих документа, посвященных конкретным технологиям, которые были разработаны совместным комитетом RTCA Special Committee #205 (SC-205) и EUROCAE Working Group #71 (WG-71):[*]

- DO-330/ED-215, *Software Tool Qualification Considerations*
- DO-331/ED-218, *Model-Based Development and Verification Supplement to DO-178C and DO-278A*
- DO-332/ED-217, *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*
- DO-333/ED-216, *Formal Methods Supplement to DO-178C and DO-278A*

В главе 4 был дан краткий обзор каждого из этих документов, подготовленных этим совместным комитетом, и их общей связи с DO-178C. Каждый документ содержит более 100 страниц, поэтому детальный разбор здесь невозможен. Каждому из четырех документов посвящена отдельная глава. В главе 13 обсуждаются DO-330 и квалификация инструментов. В главе 14 кратко рассматриваются разработка и верификация на основе моделей и документ DO-331. Глава 15 посвящена DO-332 и объектно-ориентированной технологии, а также некоторым связанным с ней приемам. В главе 16 рассматриваются формальные методы и DO-333.

Поскольку каждый из этих документов опирается на DO-178C, подробно рассмотренный в главах 5-12, в части IV анализируется связь DO-330, DO-331, DO-332 и DO-333 с DO-178C и выделяются основные различия. В каждой главе дается обзор соответствующей технологии,

ключевых отличий от DO-178C и некоторых трудностей, с которыми, вероятно, придется столкнуться при применении этих руководств. Этот обзор призван дать представление о четырех документах и помочь лучше понять, как их применять.

*Номера документов EUROCAE серии ED приведены для справки. Документы EUROCAE серии ED полностью совпадают с документами RTCA серии DO-, за исключением формата страницы и добавленного французского перевода. Во всей части IV далее будут использоваться только номера документов RTCA.

Глава 13. DO-330 и квалификация программных инструментов

Сокращения

Сокращение	Расшифровка
CNS/ATM	Communication, navigation, surveillance, and air traffic management (связь, навигация, наблюдение / организация воздушного движения)
COTS	Готовое коммерческое ПО
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
FAQ	Часто задаваемые вопросы (Frequently Asked Questions)
MC/DC	Модифицированное покрытие условий и решений
MISRA	Ассоциация надежности программного обеспечения автомобильной промышленности
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
SAS	Итоговый отчет разработки ПО (Software Accomplishment Summary)
SCMP	План управления конфигурацией ПО (Software Configuration Management Plan)
SDP	План разработки ПО (Software Development Plan)
SC-205	Special Committee #205
SVP	План верификации ПО (Software Verification Plan)
SCI	Указатель конфигурации ПО (Software Configuration Index)
SECI	Указатель конфигурации среды ЖЦ ПО (Software Life Cycle Environment Configuration Index)
TOR	Эксплуатационное требование к инструменту (Tool Operational Requirement)
TQL	Уровень квалификации инструмента (Tool Qualification Level)
TQP	План квалификации инструмента (Tool Qualification Plan)
WG-71	Working Group #71

13.1 Введение

Программный инструмент – это “компьютерная программа или ее функциональная часть, используемая для помощи в разработке, преобразовании, тестировании, анализе, создании или модификации другой программы, ее данных или

ее документации” [1]. Использование инструментов в процессе жизненного цикла программного обеспечения, а также в других областях, таких как системы, программируемая аппаратура и авиационные базы данных, в последние годы резко выросло. Недавно мне довелось работать над проектом, в котором использовалось около 50 программных инструментов. Это был огромный проект с множеством специализированных потребностей, поэтому такой пример не является типичным. Однако он показывает, насколько выросло использование инструментов и что этот рост будет продолжаться. В таблице 13.1 приведены некоторые распространенные способы использования инструментов в жизненном цикле ПО, разделенные на следующие категории: разработка, верификация и прочее.

Таблица 13.1 Примеры использования инструментов в жизненном цикле программного обеспечения

Инструменты		
Инструменты разработки	верификации	Прочие инструменты
Управление требованиями и их фиксация	Отладка	Управление проектом
Статический анализ	Управление конфигурацией	Проектирование
Анализ времени наихудшего выполнения	Моделирование	Текстовое редактирование
Верификация моделей	Сообщения о проблемах	Компиляция
Проверка соответствия стандартам кодирования	Экспертная оценка	Компоновка
Верификация трассируемости	Автоматическая генерация кода	Анализ структурного покрытия
Генерация файлов конфигурации	Автоматическое тестирование	Эмуляция
Моделирование	Автоматическая генерация тестов	Верификация файлов конфигурации
Формальные методы		

Инструменты не могут заменить человеческий мозг. Однако они могут предотвращать ошибки и выявлять те ошибки, которые человек может внести или не заметить. Инструменты помогают инженерам лучше выполнять свою работу и позволяют им сосредоточиться на более сложных задачах, требующих инженерных навыков и суждения.

Некоторые инженеры сопротивляются использованию инструментов, тогда как другие впадают в другую крайность и применяют инструменты вообще для всего. При использовании инструментов необходимо найти разумную середину, чтобы успешно разрабатывать про-

граммное обеспечение, выполняющее свою предназначенную функцию.

В 2004 году мне выпала честь возглавить форум *Software Tools Forum*. Форум был совместно организован Embry-Riddle Aeronautical University и Федеральным управлением гражданской авиации США (Federal Aviation Administration, FAA). Целью этой работы была оценка состояния инструментов в авиации и выявление вопросов, которые необходимо решить, чтобы отрасль могла безопасно получить выгоду от использования инструментов. В форуме приняли участие примерно 150 представителей промышленности, государства и академической среды. Целями форума были: (1) обмен информацией о программных инструментах, используемых в авиационных проектах, (2) обсуждение уроков, извлеченных к тому моменту из применения программных инструментов в авиации, (3) обсуждение проблем, связанных с инструментами в системах, критичных к безопасности, и (4) рассмотрение дальнейших шагов.

В конце форума был проведен мозговой штурм, чтобы определить некоторые из наиболее значимых вопросов, связанных с эффективным использованием инструментов в авиации. В ходе этого мозгового штурма было выявлено шесть категорий потребностей, которые затем были расставлены отраслью по приоритету. Первые пять категорий являются конкретными, а последняя представляет собой сборную категорию для прочих вопросов по инструментам. Эти шесть категорий перечислены здесь в порядке приоритета (пункты 1-3 получили значительно больший вес, чем пункты 4-6):

1. Необходимо пересмотреть критерии квалификации инструментов разработки.
2. Необходимо установить критерии для инструментов разработки на основе моделей.
3. Необходимо разработать критерии, позволяющие переносить зачет квалификации инструмента с одного проекта на другой.
4. Необходимо разработать и документировать различные подходы к использованию и квалификации генераторов автокода.
5. Инструменты интеграции создают новые проблемы, которые необходимо решать.
6. Необходимо решить ряд прочих вопросов, связанных с инструментами.

Вскоре после этого мероприятия по инструментам был создан совместный комитет RTCA Special Committee #205 (SC-205) и EUROCAE Working Group #71 (WG-71), чтобы обновить DO-178C и подготовить необходимое руководство в ряде технических областей. В качестве входных данных для комитета использовались рекомендации *Software Tools Forum* и других источников. Подгруппа этого комитета по квалификации инструментов[*] серьезно отнеслась к этим вопросам и занялась документированием руководства, которое должно было их закрыть. Результатом стали обновление раздела 12.2 документа DO-178C и выпуск DO-330

под названием *Software Tool Qualification Considerations*. В течение 6,5 лет, которые потребовались для завершения работы комитета, мне также довелось рассматривать и утверждать десятки программных инструментов и их квалификационные данные. Тема квалификации программных инструментов мне по-настоящему близка.

В этой главе дается обзор раздела 12.2 DO-178C и документа DO-330, чтобы объяснить, когда инструмент нужно квалифицировать, какой уровень квалификации требуется и как квалифицировать инструмент. Объем DO-330 составляет 128 страниц, поэтому здесь дается только обзор. В этой главе внимание сосредоточено на наиболее критичных аспектах DO-330 для разработчиков инструментов и их пользователей. Также обсуждаются различия между DO-178B и обновленным руководством, поскольку сегодня используется множество инструментов, квалифицированных по критериям DO-178B. Кроме того, рассматриваются некоторые специальные темы, связанные с квалификацией инструментов, а также потенциальные подводные камни, которых следует избегать при квалификации или использовании квалифицированных инструментов.

13.2 Определение необходимости и уровня квалификации инструмента (раздел 12.2 DO-178C)

В таблице 13.1 приведены распространенные способы использования инструментов в жизненном цикле программного обеспечения. Некоторые из этих инструментов могут требовать квалификации, а некоторые – нет. В этом разделе рассматривается раздел 12.2 DO-178C, чтобы ответить на следующие вопросы:

- Что такое квалификация инструмента?
- Когда она требуется?
- До какого уровня должен быть квалифицирован инструмент?
- Чем отличается руководство по квалификации инструментов в DO-178B и DO-178C?

Квалификация инструмента – это процесс, используемый для получения сертификационного зачета в отношении программного инструмента, выход которого не верифицируется, когда этот инструмент устраняет, сокращает или автоматизирует процессы, требуемые DO-178C. Квалификация инструмента предоставляется совместно с одобрением программного обеспечения, использующего этот инструмент; отдельным одобрением она не является. Процесс квалификации инструмента обеспечивает уверенность в функциональности инструмента; эта уверенность по меньшей мере эквивалентна тому процессу или процессам, которые устраняются, сокращаются или автоматизируются [2]. Степень строгости, необходимая для квалификации инструмента, “зависит от потенциального влияния ошибки инструмента на системную безопасность и от общего способа использования инструмента в

процессе жизненного цикла ПО. Чем выше риск того, что ошибка инструмента неблагоприятно повлияет на системную безопасность, тем выше требуемая строгость квалификации инструмента” [2]. На рисунке 13.1 графически показан процесс определения того, нужно ли квалифицировать инструмент и до какого уровня.

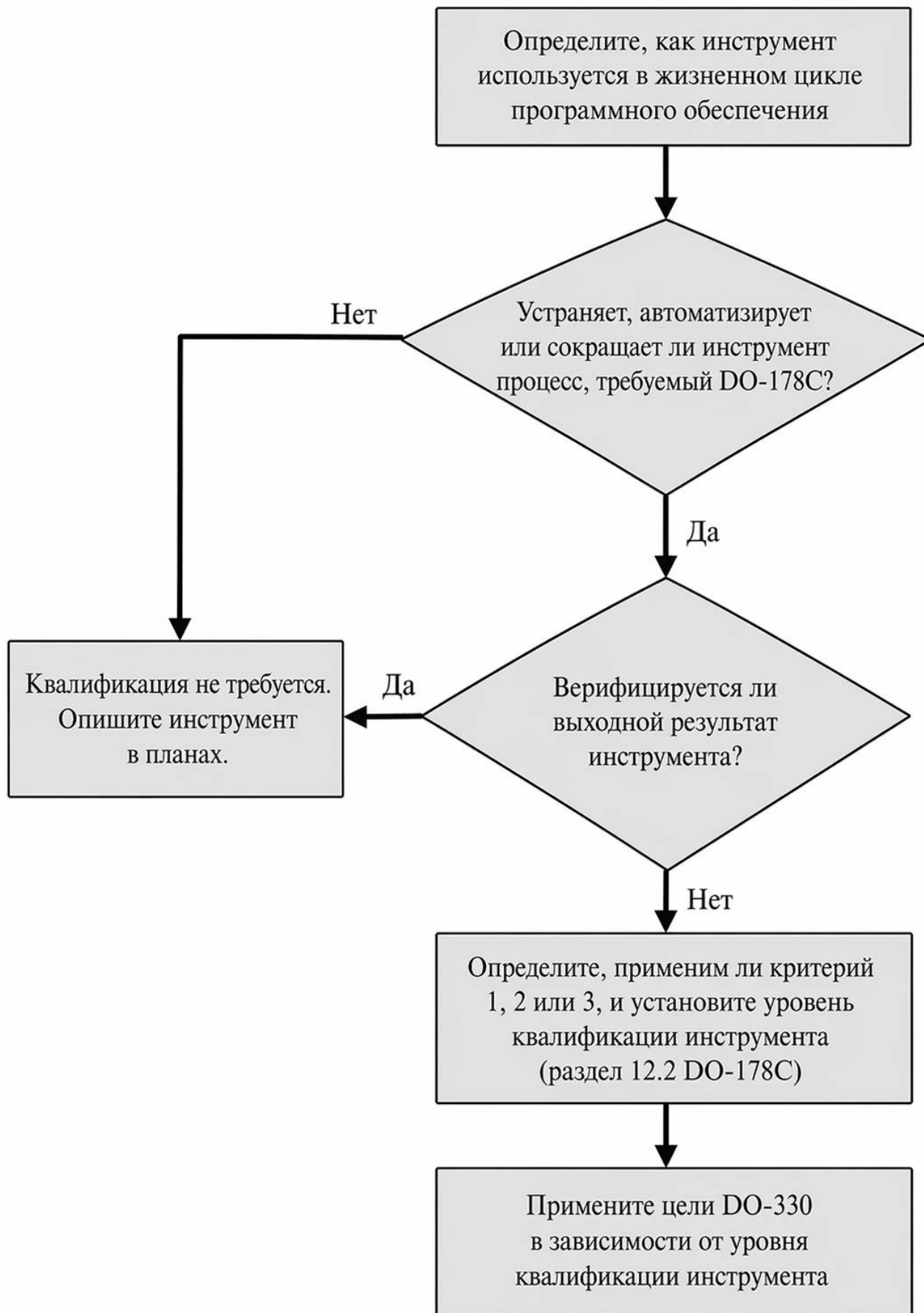


Рисунок 13.1 Определение необходимости и уровня квалификации инструмента.

DO-178В определял два типа инструментов: (1) инструменты разработки ПО и (2) инструменты верификации ПО. Поскольку эти две категории часто были привязаны скорее к этапу жизненного цикла, чем к влиянию инструмента, DO-178С не использует термины *инструменты разработки* и *инструменты верификации*. Вместо этого DO-178С определяет три критерия, сфокусированные на потенциальном влиянии инструмента. В таблице 13.2 показано сопоставление категорий инструментов DO-178В и критериев инструментов DO-178С.

Таблица 13.2 Сравнение категорий инструментов DO-178В с критериями инструментов DO-178С

Категории и определения инструментов DO-178В	Критерии и определения квалификации инструментов DO-178С
<i>Инструмент разработки</i> : инструмент, выход которого является частью бортового ПО и поэтому может внести ошибки.	<i>Критерий 1</i> : инструмент, выход которого является частью результирующего ПО и поэтому может внести ошибку.
<i>Инструмент верификации</i> : инструмент, который не может внести ошибки, но может не обнаружить их.	<i>Критерий 2</i> : инструмент, который автоматизирует процесс или процессы верификации и поэтому может не обнаружить ошибку, а его выход используется для обоснования устранения или сокращения:- процесса или процессов верификации, отличных от автоматизируемых этим инструментом; или- процесса или процессов разработки, которые могут повлиять на бортовое ПО.
	<i>Критерий 3</i> : инструмент, который в пределах своего предполагаемого использования может не обнаружить ошибку.

Источники: RTCA DO-178С, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Washington, DC, December 2011; RTCA DO-178В, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Washington, DC, December 1992.

Процесс квалификации инструментов *критерия 1* по DO-178С очень похож на процесс для инструментов *разработки* в DO-178В. Примеры инструментов *критерия 1* включают генераторы автокода, генераторы файлов конфигурации, компиляторы, компоновщики,

инструменты управления требованиями, инструменты проектирования и инструменты моделирования. Аналогично, процесс квалификации инструментов *критерия 3* по DO-178C примерно соответствует процессу для *инструментов верификации* в DO-178B. Примеры таких инструментов включают генераторы тестовых примеров, автоматизированные тестовые инструменты, инструменты структурного покрытия, инструменты анализа связности данных и статические анализаторы кода. Основное отличие между DO-178B и DO-178C состоит в том, что DO-178C вводит третью категорию инструментов: инструмент *критерия 2*. Потребность в этой специальной категории в первую очередь возникла при рассмотрении будущих инструментов, которые могут принимать более критичные решения, чем сегодняшние инструменты верификации, но не оказывают такого влияния на результирующее ПО, как сегодняшние инструменты разработки. Некоторые инструменты формальных методов попадают в эту категорию. Например, один инструмент доказательства может использоваться для автоматизации части шагов верификации исходного кода и для сокращения объема необходимого тестирования. Любое из этих действий по отдельности сделало бы его инструментом *критерия 3*, но их сочетание переводит его в область инструментов *критерия 2*. В такой ситуации инструмент доказательства используется для верификации процесса или процессов, отличных от того процесса, который автоматизируется самим инструментом. Другой пример инструмента *критерия 2* – статический анализатор кода, который используется для замены рассмотрения исходного кода как шага верификации и для сокращения проектных механизмов обнаружения переполнения как шага разработки. Такой инструмент выполняет некоторую верификацию, но также сокращает процесс разработки ПО [1]. Я склонна воспринимать инструменты *критерия 2* как *суперинструменты верификации*. С помощью одного инструмента можно закрыть цели из нескольких таблиц приложения A DO-178C. Традиционные *инструменты верификации* обычно лишь автоматизируют цели из одной таблицы приложения A DO-178C, например A-6 или A-7, а такие *суперинструменты верификации* могут удовлетворять целям из нескольких таблиц, например от A-5 до A-7. Я не ожидаю, что инструментов *критерия 2* будет много; они действительно должны быть исключением, а не правилом. Однако этот критерий был включен в DO-330, чтобы обеспечить достаточную видимость проектирования инструмента в случаях, когда один инструмент выполняет несколько назначений.

На основе трех критериев и уровня программного обеспечения, которое поддерживает инструмент, назначается уровень квалификации инструмента (Tool Qualification Level, TQL). TQL определяет степень строгости, требуемую в процессе квалификации. Всего существует пять уровней TQL; TQL-1 требует наибольшей строгости, а TQL-5 – наименьшей. В таблице 13.3 показан TQL для каждого критерия инструмента и уровня ПО. В таблице 13.4 показана корреляция между подходом DO-178B к уровням ПО и TQL в DO-178C. Замысел

DO-178C и DO-330 состоит в том, чтобы сделать критерии квалификации инструментов яснее, чем это было в DO-178B, а не изменить сами требования. Поэтому в большинстве случаев инструменты, квалифицированные по DO-178B, будут приемлемы и по DO-178C.

Таблица 13.3 Уровень квалификации инструмента

Уровень программного обеспечения			
	Критерий 1	Критерий 2	Критерий 3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

Источник: RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc., Washington, DC, December 2011. Таблица 12-2 из DO-178C использована с разрешения RTCA, Inc.

Таблица 13.4 Корреляция между уровнями DO-178B и DO-178C

Тип квалификации инструмента по DO-178B	Уровень программного обеспечения по DO-178B Уровень TQL по DO-178C
Разработка	ATQL-1
Разработка	BTQL-2
Разработка	CTQL-3
Разработка	DTQL-4
Верификация	Все TQL-4a или TQL-5b

Источник: RTCA DO-330, *Software Tool Qualification Considerations*, RTCA, Inc., Washington, DC, December 2011. Таблица D-1 из DO-330 использована с разрешения RTCA, Inc.

aTQL-4 применяется к инструментам критерия 2, используемым для программного обеспечения уровня A и B.

bTQL-5 применяется к инструментам критерия 2, используемым для программного обеспечения уровня C и D, а также к инструментам критерия 3.

13.3 Квалификация инструмента (обзор DO-330)

После того как с помощью раздела 12.2 DO-178C установлен уровень квалификации инструмента, DO-330 предоставляет руководство по квалификации инструмента. В этом разделе объясняется, зачем нужен DO-330 и

как устроен процесс квалификации инструмента по DO-330.

13.3.1 Необходимость DO-330

За последние несколько лет число и типы инструментов выросли. Кроме того, инструменты часто разрабатываются сторонними поставщиками, которые могут слабо понимать DO-178C или вовсе его не понимать. DO-330 был разработан, чтобы предоставить руководство, ориентированное именно на инструменты, как для разработчиков инструментов, так и для их пользователей. Документ сосредоточен на инструментах, используемых в жизненном цикле программного обеспечения. Однако он может применяться и к другим областям, таким как программируемая аппаратура, базы данных и системы. То, как документ будет применяться к конкретной области, определяется руководством этой области, например DO-178C, DO-254 (KT-254) [3], DO-200A [4].

Комитет довольно оживленно спорил о том, нужен ли вообще отдельный документ по квалификации инструментов. У подготовки такого документа есть преимущества и недостатки. В таблице 13.5 обобщены некоторые доводы за и против. Эти и другие соображения учитывались в ходе обсуждений комитета. В итоге было решено, что преимущества отдельного документа перевешивают последствия и что отдельный документ является наилучшим долгосрочным решением для авиационной отрасли в целом.

Таблица 13.5 Плюсы и минусы отдельного документа по квалификации инструментов

Плюсы отдельного документа	Минусы отдельного документа
Во многих проектах фактически строк кода в инструментах больше, чем в бортовом программном обеспечении; поэтому требуется специальное руководство.	Существует значительная избыточность между документом по квалификации инструментов (DO-330) и DO-178C.
Инструменты образуют отдельную область по отношению к бортовому программному обеспечению. Отдельный документ позволяет напрямую решать потребности, характерные именно для инструментов, а не оставаться на уровне общих рассуждений. Это предотвращает неверные толкования.	Руководство по квалификации инструментов трудно применять, когда инструменты разработаны с использованием нетрадиционных технологий, таких как разработка на основе моделей, объектно-ориентированная технология или формальные методы, то есть дополнения объясняют, как они применяются к DO-178C и DO-278A, но не к DO-330.

Плюсы отдельного документа	Минусы отдельного документа
Отдельный документ дает руководство специально для разработчиков инструментов, которые могут почти ничего не знать о DO-178C.	Еще один документ создает сложности при сопровождении руководства, особенно там, где есть перекрытие между документом и DO-178C.
Поскольку и в других областях, таких как программируемая аппаратура, системы и авиационные базы данных, также используются инструменты, отдельный документ может быть полезен не только для области программного обеспечения.	Дополнительный документ может привести к дополнительной работе для разработчиков. Особенно для тех, кто квалифицирует только инструменты низших уровней, то есть инструменты верификации в DO-178B или TQL-5 в DO-178C.

При разработке DO-330 комитет держал в уме следующие цели:

- Сохранить подход DO-178B для традиционных инструментов верификации, насколько это возможно.
- Разработать подход, который будет поддерживать новые инструментальные технологии.
- Предусмотреть подход, позволяющий повторно использовать зачет квалификации инструмента в нескольких проектах.
- Определить общие роли пользователя инструмента и разработчика инструмента, чтобы учесть интеграцию инструмента в среду разработки, поддержать повторное использование и помочь с интеграцией готовых коммерческих продуктов.
- Разработать подход, который был бы ясным, но в то же время гибким для разработчиков и пользователей инструментов.
- Разработать подход, основанный на целях, чтобы помочь повторному использованию и гибкости.
- Предусмотреть подход, который может быть принят в нескольких областях.

13.3.2 Процесс квалификации инструмента по DO-330

После того как определены необходимость квалификации и применимый уровень квалификации инструмента с использованием раздела 12.2 DO-178C, DO-330 предоставляет руководство и цели для квалификации инструмента. DO-330 организован похоже на DO-178C. Разработка инструмента проходит через жизненный цикл, так же как и бортовое программное обеспечение. Процессы жизненного цикла инструмента включают планирование, тре-

бования, проектирование, верификацию, управление конфигурацией, обеспечение качества и взаимодействие по вопросам квалификации. В таблице 13.6 рядом показаны оглавления DO-178C и DO-330, что иллюстрирует, что DO-330 организован очень похоже на DO-178C. В качестве основы для DO-330 был использован DO-178C, а затем его содержание было изменено так, чтобы сделать документ ориентированным на инструменты.

Таблица 13.6 Сравнение оглавлений DO-178C и DO-330

Оглавление DO-178C [2]	Оглавление DO-330 [1]
1. Введение	1. Введение
2. Системные аспекты, относящиеся к разработке программного обеспечения	2. Назначение квалификации инструментов
3. Жизненный цикл ПО	3. Характеристики квалификации инструментов
4. Процесс планирования ПО	4. Процесс планирования квалификации инструмента
5. Процессы разработки ПО	5. Жизненный цикл и процессы разработки инструмента
6. Процессы верификации ПО	6. Процессы верификации инструмента
7. Процесс управления конфигурацией	7. Процесс управления конфигурацией инструмента
8. Процесс обеспечения качества ПО	8. Процесс обеспечения качества инструмента
9. Процесс взаимодействия по вопросам сертификации	9. Процесс взаимодействия по вопросам квалификации инструмента
10. Обзор процесса сертификации	10. Данные квалификации инструмента
11. Данные жизненного цикла ПО	11. Дополнительные соображения по квалификации инструмента
12. Дополнительные соображения	Приложение А – Цели квалификации инструмента
Приложение А – Цели процессов и выходные данные по уровням ПО	Приложение В – Сокращения и глоссарий терминов
Приложение В – Сокращения и глоссарий терминов	Приложение С – Часто задаваемые вопросы, связанные с квалификацией инструментов для всех областей

Оглавление DO-178С [2]	Оглавление DO-330 [1]
Приложение С – Предыстория документа DO-178	Приложение D – Часто задаваемые вопросы, связанные с квалификацией инструментов для областей бортового ПО и ПО CNS/АТМ (связь, навигация, наблюдение / организация воздушного движения)
Приложение D – Состав комитета	

Поскольку DO-178С и DO-330 похожи, а DO-178С был подробно рассмотрен в главах 5-12, в этом разделе подчеркиваются 18 основных различий между DO-178С и DO-330.

Различие 1: разные области применения. Как уже упоминалось, DO-178С относится к области бортового программного обеспечения, тогда как DO-330 относится к инструментам, которые могут использоваться в жизненном цикле ПО. Кроме того, DO-330 может использоваться для квалификации инструментов, применяемых и в других областях, например в ПО для связи, навигации, наблюдения и организации воздушного движения (Communication, Navigation, Surveillance, and Air Traffic Management, CNS/АТМ), системах и электронной аппаратуре.

Различие 2: различаются вводные разделы. DO-178С в разделе 2 содержит обзор системного процесса и уровней программного обеспечения. DO-330 в разделах 2 и 3 объясняет назначение и характеристики квалификации инструментов. Информация в разделах 2 и 3 DO-330 в целом похожа на то, что уже обсуждалось в этой главе.

Различие 3: в DO-330 объединены разделы о процессах жизненного цикла и планировании. В DO-178С есть отдельные разделы, посвященные жизненному циклу программного обеспечения и планированию, это разделы 3 и 4. Однако в DO-330 эти темы объединены в один раздел, раздел 4, поскольку жизненный цикл играет значительную роль в процессе планирования.

Различие 4: в DO-330 к процессам жизненного цикла и данным добавляется слово tool (инструмент). Чтобы отличать процесс разработки инструмента от процессов разработки бортового программного обеспечения, там, где это уместно, в термин вводится слово *tool*. Например, термин *source code* в DO-178С соответствует термину *tool source code* в DO-330, то есть исходному коду инструмента.

Примечание переводчика. В российской практике для этой пары документов удобно держать в уме соответствие: DO-178С – КТ-178С, DO-330 – Р-330.

Различие 5: DO-330 расширяет перечень сведений, требуемых в Плане сертификации ПО и Итоговом отчёте разработки ПО при использовании квалифицированных инструментов. Разделы 11.1, 11.20 DO-178C определяют ожидаемое содержимое Плана сертификации ПО (Plan for Software Aspects of Certification, PSAC) и Итогового отчёта разработки ПО (Software Accomplishment Summary, SAS). Разделы 10.1.1, 10.1.16 DO-330 определяют дополнительные сведения, которые следует включать в План сертификации ПО и Итоговый отчёт разработки ПО, когда в процессах жизненного цикла программного обеспечения предполагается использовать квалифицированные инструменты. Согласно разделу 10.1.1 DO-330, для каждого квалифицированного инструмента, который будет использоваться, в План сертификации ПО следует добавить следующую информацию [1]:

- Указать инструмент и его предполагаемое использование в процессе жизненного цикла программного обеспечения. Объяснить, какой зачет запрашивается по этому инструменту, то есть какие процессы или цели он устраняет, сокращает или автоматизирует.
- Обосновать зрелость технологии, автоматизируемой инструментом. Например, если инструмент используется для выполнения формальных доказательств, необходимо убедиться в зрелости самого подхода в целом до его автоматизации.
- Предложить уровень квалификации инструмента и привести обоснование, почему именно этот уровень является достаточным. Указать разработчика инструмента или источник инструмента. Объяснить распределение ролей и обязанностей в квалификации инструмента. В частности, объяснить, что делает пользователь инструмента, что делает разработчик инструмента и кто именно выполняет какие цели DO-330.
- Объяснить процессы разработки эксплуатационных требований к инструменту, интеграции инструмента, а также эксплуатационной валидации и верификации инструмента. Указать предполагаемую эксплуатационную среду инструмента и подтвердить, что она представительна для среды, использовавшейся при верификации инструмента. Объяснить любые дополнительные соображения по инструменту, например повторное использование, применение готовых коммерческих продуктов или опыт эксплуатации.
- Дать ссылку на План квалификации инструмента, где можно найти дополнительные сведения о деталях квалификации инструмента для TQL-1 – TQL-4. Для TQL-5 план квалификации инструмента не требуется.

Раздел 10.1.16 DO-330 также объясняет, какую дополнительную информацию следует включать в Итоговый отчёт разработки ПО при использовании квалифицированных инструментов в жизненном цикле программного обеспечения. В таблице 13.7 обобщены дополнительные пункты для уровней TQL-1-TQL-4 и TQL-5.

Таблица 13.7 Дополнительные включения в Итоговый отчёт разработки ПО при использовании квалифицированных инструментов

TQL-1 - TQL-4	TQL-5
Идентификация инструмента (конкретное обозначение или иной идентификатор).	Идентификация инструмента (конкретное обозначение или иной идентификатор).
Подробности запрашиваемого сертификационного зачета для инструмента, то есть объяснение того, какие процессы и/или цели он устраняет, сокращает или автоматизирует.	Подробности запрашиваемого сертификационного зачета для инструмента, то есть объяснение того, какие процессы и/или цели он устраняет, сокращает или автоматизирует.
Ссылка на Итоговый отчёт по квалификации инструмента, где можно найти дополнительную информацию о деталях квалификации инструмента.	Перечень или ссылка на данные квалификации инструмента, включая сведения о версии.
Заявление, подтверждающее, что разработка инструмента, верификация и интегральные процессы соответствуют планам по инструменту, включая мероприятия пользователя инструмента.	
Сводка по всем открытым сообщениям о проблемах инструмента вместе с анализом, подтверждающим, что поведение инструмента соответствует эксплуатационным требованиям к инструменту.	
Сводка по любым отклонениям в использовании инструмента по сравнению с тем, что было описано в Плане сертификации ПО, если это применимо.	

Источник: RTCA DO-330, *Software Tool Qualification Considerations*, RTCA, Inc., Washington, DC, December 2011.

Различие 6: DO-330 различает эксплуатацию инструмента и разработку инструмента.

DO-178B не проводил такого различия, и это приводило к путанице и ограничивало повторное использование. Чтобы поддержать повторное использование инструмента и обеспечить ясность, DO-330 дает руководство отдельно для разработки и верификации инструмента, выполняемых разработчиком инструмента, и для эксплуатации инструмента, выполняемой пользователем инструмента. При этом стандарт оставляет проекту гибкость в распределении ролей и обязанностей с учетом конкретной ситуации. В общем случае цели таблиц T-0 и T-10 приложения A DO-330 в первую очередь относятся к пользователю или оператору инструмента, тогда как цели таблиц T-1 – T-9 приложения A DO-330 в основном относятся к разработчику инструмента.

Различие 7: руководство DO-330 требует наличия эксплуатационных требований для документирования потребностей пользователя. В DO-178C системные требования задают требования высокого уровня к ПО. Однако для инструмента системных требований не существует, вместо них требования к инструменту определяются эксплуатационными требованиями к инструменту. Эти эксплуатационные требования определяют, как инструмент используется в конкретном процессе жизненного цикла программного обеспечения. Разные пользователи общего инструмента могут применять его по-разному, поэтому эксплуатационные требования к инструменту зависят от пользователя. Документ с эксплуатационными требованиями к инструменту включает описание следующего [1]:

- Контекст использования инструмента в жизненном цикле программного обеспечения, интерфейсы с другими инструментами и интеграцию выходных файлов инструмента в бортовое ПО. Эксплуатационные среды, в которых инструмент будет использоваться.
- Входные данные инструмента, включая формат файлов, описание языка и так далее. Формат и краткое описание содержимого выходных файлов инструмента. Функциональные требования к инструменту.
- Требования к аномальным режимам активации или к несогласованным входным данным, которые должны обнаруживаться инструментом. Это требуется для уровней квалификации 1-4, но не требуется для уровня квалификации 5.
- Требования к производительности, то есть к поведению выходных данных инструмента.
- Информация для пользователя, объясняющая, как правильно использовать инструмент. Описание того, как запускать инструмент, включая выбранные опции, значения параметров, командную строку и так далее.

Различие 8: типы требований в DO-330 отличаются. DO-178C выделяет системные требования, требования высокого уровня к ПО и требования низкого уровня к ПО. DO-330

выделяет эксплуатационные требования к инструменту, находящиеся на том же базовом уровне, что и системные требования в DO-178C, требования к инструменту, имеющие тот же замысел, что и требования высокого уровня к ПО в DO-178C, и требования низкого уровня к инструменту, имеющие ту же степень детализации, что и требования низкого уровня к ПО в DO-178C. В некоторых ситуациях три уровня требований могут не понадобиться. Например, эксплуатационные требования к инструменту и требования к инструменту могут представлять собой один набор требований, либо требования к инструменту и требования низкого уровня к инструменту могут составлять один уровень требований [*]. Даже если уровни требований объединены, все применимые цели DO-330 все равно должны быть выполнены. Как и в DO-178C, требуется двунаправленная трассируемость между различными уровнями требований к инструменту.

Различие 9: DO-330 требует валидации эксплуатационных требований к инструменту. В DO-178C системные требования, поступающие в программный процесс, валидируются с использованием процесса, соответствующего ARP4754A, поэтому DO-178C не требует валидации требований. Однако эксплуатационные требования к инструменту не валидируются системным процессом, поэтому они должны быть валидированы на корректность и полноту.

Различие 10: согласно DO-330 производные требования к инструменту оцениваются с точки зрения влияния на функциональность инструмента и эксплуатационные требования к инструменту, а не на процесс оценки безопасности. Поскольку инструменты сами не летают, их влияние на процесс оценки безопасности является косвенным. Вместо этого оценивается влияние производных требований на жизненный цикл программного обеспечения, ожидаемую функциональность и эксплуатационные требования к инструменту. Как и в DO-178C, все производные требования должны быть обоснованы. Пользователи могут не знать подробностей проектирования инструмента, поэтому обоснование производных требований помогает им правильно оценить влияние на их жизненный цикл ПО. Эти обоснования следует писать с учетом интересов пользователя.

Различие 11: в DO-330 используется понятие эксплуатационной среды инструмента вместо целевого компьютера. DO-330 определяет эксплуатационную среду инструмента так: «Среда и контекст процессов жизненного цикла, в которых используется инструмент. Она включает рабочую станцию, операционную систему и внешние зависимости, например интерфейсы с другими инструментами и ручными процессами» [1]. Инструменты обычно запускаются на настольных компьютерах, а не встраиваются в авионику. Поэтому понятие *целевого компьютера*, используемое в DO-178C, плохо подходит для процесса квалификации. Поскольку инструменты нередко разрабатываются для использования в нескольких эксплуатационных средах, важно идентифицировать каждую такую среду.

Различие 12: DO-330 дает больше гибкости в подходах к структурному покрытию и делает акцент на выявлении непредусмотренной функциональности. Структурное покрытие, вероятно, было самой спорной темой во время обсуждений DO-330 в рамках работы комитета. Изначально модифицированное покрытие условий и решений (Modified Condition/Decision Coverage, MC/DC) не включалось в DO-330, поскольку сами инструменты не летают, а исторические данные показывают, что для инструментов ценность такого покрытия ограничена. Были включены покрытие решений и покрытие операторов. Однако цель по MC/DC все же добавили, вместе с целями по покрытию решений и операторов, чтобы добиться консенсуса в комитете. Согласно DO-330, покрытие операторов применяется для TQL-3; покрытие операторов и решений применяется для TQL-2; а покрытие операторов, решений и MC/DC применяется для TQL-1. Однако формулировки DO-330 подчеркивают, что назначение структурного покрытия состоит в выявлении непредусмотренной функциональности. Это открывает путь для альтернатив MC/DC, при условии что такие альтернативы достигают того же замысла, что и MC/DC. Еще одно отличие, связанное со структурным покрытием, состоит в том, что DO-330 не требует анализа трассируемости от исходного кода к объектному коду для кода инструмента уровня TQL-1, тогда как для программного обеспечения уровня А по DO-178С это требуется.

Различие 13: DO-330 рассматривает интерфейсы с внешними компонентами. Внешние компоненты определяются как «компоненты программного обеспечения инструмента, находящиеся вне контроля разработчика инструмента. Примерами могут быть примитивные функции, предоставляемые операционной системой или библиотекой времени выполнения компилятора, либо функции, предоставляемые библиотекой готового коммерческого продукта или библиотекой с открытым исходным кодом» [1]. DO-330 включает цели и мероприятия для выполнения следующего [1]:

- Идентифицировать интерфейсы с внешними компонентами в проекте (раздел 5.2.2.2.g DO-330).
- Проверять корректность и полноту интерфейсов (раздел 6.1.3.3.e DO-330).
- Обеспечивать корректную интеграцию интерфейсов к внешним компонентам (раздел 6.1.3.5 DO-330).
- Обеспечивать, чтобы интерфейсы к внешним компонентам были задействованы при тестировании, основанном на требованиях (раздел 6.1.4.3.2 DO-330).

Различие 14: DO-330 требует отчет об установке инструмента. Отчет об установке инструмента формируется на этапе эксплуатационной интеграции для подтверждения того, что инструмент интегрирован в свою эксплуатационную среду. В отчете об установке указываются: (1) конфигурация эксплуатационной среды, (2) версия исполняемого объектного

кода инструмента и любых поддерживающих файлов конфигурации, (3) все внешние компоненты и (4) способ запуска инструмента [1].

Различие 15: данные жизненного цикла в DO-330 специфичны для инструментов. Большая часть данных жизненного цикла инструмента, требуемых DO-330, похожа на то, что требуется DO-178C для бортового программного обеспечения или DO-278A для ПО связи, навигации, наблюдения и организации воздушного движения. Однако названия документов ориентированы именно на инструменты, а рекомендуемое содержимое сосредоточено на нуждах квалификации инструмента. Раздел 10 DO-330 описывает данные жизненного цикла инструмента, а таблицы целей приложения A DO-330 показывают, какие данные требуются для каждого TQL. Раздел 10 DO-330 делит данные на три категории, каждая из которых описана в отдельном подразделе. Эти три категории и соответствующие данные показаны в таблице 13.8.

Таблица 13.8 Данные квалификации инструмента в DO-330

Категория данных	Содержимое
Данные для процесса взаимодействия по квалификации инструмента и других интегральных процессов, раздел 10.1 DO-330	План сертификации ПО с инструментно-специфической информацией, План квалификации инструмента, план разработки инструмента, план верификации инструмента, план управления конфигурацией инструмента, план гарантии качества инструмента, стандарты требований, проектирования и кода инструмента, указатель конфигурации среды жизненного цикла инструмента, указатель конфигурации инструмента, сообщения о проблемах инструмента, записи управления конфигурацией инструмента, записи гарантии качества инструмента, итоговый отчёт разработки инструмента, итоговый отчёт разработки ПО с инструментно-специфической информацией и указатель конфигурации среды ЖЦ ПО с инструментно-специфической информацией.
Данные, создаваемые в ходе разработки и верификации инструмента, раздел 10.2 DO-330	Требования к инструменту, описание проектирования инструмента, исходный код инструмента, исполняемый объектный код инструмента, примеры и процедуры верификации инструмента, результаты верификации инструмента, данные трассировки.

Категория данных	Содержимое
Данные для поддержки эксплуатационного одобрения инструмента, раздел 10.3 DO-330	Эксплуатационные требования к инструменту, отчет об установке инструмента, примеры и процедуры эксплуатационной верификации и валидации инструмента, результаты эксплуатационной верификации и валидации инструмента.

Различие 16: таблицы целей DO-330 в ряде мест отличаются от таблиц целей DO-178C. Таблицы целей в приложении А DO-330 похожи на таблицы из приложения А DO-178C. Однако между ними есть некоторые различия. Одно заметное отличие состоит в том, что таблицы приложения А в DO-330 нумеруются как Т-х, а не А-х, чтобы отличать их от DO-178C. Еще одно общее отличие состоит в том, что в DO-330 имеется 11 таблиц приложения А, а не 10, как в DO-178C. Названия таблиц приложения А в DO-330 приведены ниже, а затем следует краткое описание основных отличий между таблицами приложения А в DO-330 и DO-178C:

- *Таблица Т-0: Эксплуатационные процессы инструмента*
- *Таблица Т-1: Процесс планирования инструмента*
- *Таблица Т-2: Процессы разработки инструмента*
- *Таблица Т-3: Верификация выходных данных процессов требований к инструменту*
- *Таблица Т-4: Верификация выходных данных процесса проектирования инструмента*
- *Таблица Т-5: Верификация выходных данных процессов кодирования и интеграции инструмента*
- *Таблица Т-6: Тестирование выходных данных процесса интеграции*
- *Таблица Т-7: Верификация выходных данных тестирования инструмента*
- *Таблица Т-8: Процесс управления конфигурацией инструмента*
- *Таблица Т-9: Процесс обеспечения качества инструмента*
- *Таблица Т-10: Процесс взаимодействия по квалификации инструмента*

Таблица Т-0 является специальной таблицей для инструментов в DO-330 и не имеет эквивалента в DO-178C. Она включает семь целей для четырех процессов инструмента, связанных с эксплуатацией инструмента, то есть с точки зрения пользователя. Эти процессы и цели кратко показаны в таблице 13.9.

Таблица 13.9 Краткая сводка процессов и целей таблицы T-0 из DO-330

Процесс из таблицы T-0	Цель(и) из таблицы T-0
Планирование	1. «Необходимость квалификации инструмента установлена.»
Разработка эксплуатационных требований к инструменту	2. «Эксплуатационные требования к инструменту определены.»
Эксплуатационная валидация и верификация инструмента	3. «Исполняемый объектный код инструмента установлен в эксплуатационной среде инструмента.»
Эксплуатационная интеграция инструмента	4. «Эксплуатационные требования к инструменту полны, точны, проверяемы и непротиворечивы.»
Эксплуатационная интеграция инструмента	5. «Эксплуатация инструмента соответствует эксплуатационным требованиям к инструменту.»
Эксплуатационная интеграция инструмента	6. «Эксплуатационные требования к инструменту достаточны и корректны.»
Эксплуатационная интеграция инструмента	7. «Потребности процессов жизненного цикла программного обеспечения удовлетворяются инструментом.»

Источник: RTCA DO-330, *Software Tool Qualification Considerations*, RTCA, Inc., Washington, DC, December 2011.

Таблицы T-1 – T-10 в DO-330 похожи на таблицы A-1 – A-9 в DO-178C, за следующими существенными исключениями:

- Цель 8 таблицы T-2 в DO-330 гласит: «Инструмент установлен в среде(ах) верификации инструмента» [1]. Эта цель характерна именно для инструментов, поскольку среда верификации инструмента может отличаться от эксплуатационной среды. Кроме того, если сред верификации несколько, инструмент придется интегрировать в каждую из них в ходе квалификации.
- Цель 3 таблицы T-3 в DO-330 гласит: «Требования к совместимости с эксплуатационной средой инструмента определены» [1]. Эта цель ориентирована на совместимость с эксплуатационной средой, а не со средой целевого компьютера, о чем уже говорилось в различии 11.
- Цель 4 таблицы T-3 в DO-330 гласит: «Требования к инструменту определяют пове-

дение инструмента в ответ на условия ошибки» [1]. Эта цель характерна именно для инструментов и сосредоточена на том, чтобы инструмент был устойчивым и предсказуемо реагировал на ошибки.

- Цель 5 таблицы T-3 в DO-330 гласит: «Требования к инструменту определяют инструкции для пользователя и сообщения об ошибках» [1]. И это тоже специальная цель для инструментов. Она показывает, что в DO-330 учитывается точка зрения пользователя.
- Цель 11 таблицы T-4 в DO-330 гласит: «Интерфейс внешнего компонента определен корректно и полностью» [1]. Эта специальная для инструментов цель предназначена для верификации интерфейса внешнего компонента, обсуждавшегося ранее; см. различие 13.
- Цель 5 таблицы T-7 в DO-330 гласит: «Выполнен анализ тестирования внешних компонентов, основанного на требованиях» [1]. В DO-178C эквивалентной цели нет. Как отмечалось ранее в различии 13, эта цель предназначена для подтверждения того, что тестирование, основанное на требованиях, действительно задействовало интерфейсы инструмента с внешними компонентами.
- Таблица T-10 в DO-330, по всем своим целям, сосредоточена на *квалификации* инструмента, а не на *сертификации*.

Различие 17: TQL-5 в DO-330 уточняет то, что требовалось для инструментов верификации в DO-178B. DO-330 задумывался так, чтобы требовать для квалификации инструментов TQL-5 ту же степень строгости, какая требовалась для инструментов верификации в DO-178B. Однако критерии DO-330 уточняют ожидания DO-178B, чтобы обеспечить соответствие и поддержать повторное использование инструмента. Часто задаваемый вопрос D.6 в DO-330 так суммирует различия между инструментом верификации из DO-178B и TQL-5 из DO-330:

Этот документ [DO-330] дает более точное и полное руководство для инструментов уровня TQL-5, чем DO-178B (и, следовательно, DO-278) давал для инструментов верификации. Замысел не в том, чтобы требовать больше мероприятий или больше данных, например квалификация не требует никаких данных из процесса разработки инструмента. Однако он уточняет содержание эксплуатационных требований к инструменту, соответствие инструмента потребностям процессов бортового программного обеспечения или ПО связи, навигации, наблюдения и организации воздушного движения, а также цели других интегральных процессов, применимых к уровню квалификации 5 [1].[*]

Некоторые из уточнений в DO-330 для TQL-5, которых не было в критериях для инструментов верификации в DO-178B, можно суммировать так:

- DO-330 определяет конкретные цели для TQL-5, как и для всех TQL. В DO-178B этого не было, то есть для квалификации инструментов верификации в DO-178B отдельные цели не определялись.
- DO-330 дает дополнительные разъяснения о том, что ожидается в эксплуатационных требованиях к инструменту. При этом DO-330 разделяет эксплуатационные требования к инструменту и требования к инструменту.
- DO-330 добавляет цель по интеграции, чтобы квалификация выполнялась в рамках конкретной эксплуатационной среды.
- DO-330 включает цели, обеспечивающие валидацию эксплуатационных требований к инструменту.

Различие 18: в DO-330 есть часто задаваемые вопросы в приложениях C и D. Часто задаваемые вопросы (Frequently Asked Questions, FAQ) и дискуссионные материалы по DO-178C включены в DO-248C. Однако часто задаваемые вопросы, относящиеся к инструментам, включены в приложения DO-330. Приложение C содержит вопросы, независимые от домена. Приложение D содержит вопросы, специфичные для доменов DO-178C и DO-278A. В таблице 13.10 кратко описан каждый такой вопрос.

Таблица 13.10 Часто задаваемые вопросы из приложений C и D DO-330

Номер вопроса	Тема
C.1	Применение DO-330 в других доменах
C.2	Определение TQL, если квалификация инструмента выполняется в сочетании с несколькими приложениями
C.3	Изменение роли инструмента
C.4	Использование DO-330 для квалификации нескольких инструментов
C.5	Использование и квалификация инструментов, если инструмент разделен между несколькими вычислительными платформами
C.6	Использование повторно применяемых компонентов для целей квалификации инструмента

Номер вопроса	Тема
C.7	Учитываются ли мероприятия, выполнявшиеся до квалификации инструмента, для этой квалификации
D.1	Как квалифицировать инструмент, когда квалификация выполняется совместно с несколькими пользователями
D.2	Нужно ли учитывать квалифицированный инструмент в анализе покрытия по структуре кода бортового программного обеспечения
D.3	Возможен ли TQL-5 для инструмента, который заменяет ручную верификацию
D.4	Как следует квалифицировать инструмент, работающий с неисполняемыми данными при разработке требований
D.5	Какой уровень эксплуатационных требований к инструменту необходим для инструмента уровня квалификации 5
D.6	Можно ли использовать существующую квалификацию инструмента верификации по DO-178B как средство соответствия DO-330 для TQL-5
D.7	Каков объем верификации для эксплуатационных требований к инструменту

Источник: RTCA DO-330, Software Tool Qualification Considerations, RTCA, Inc., Washington, DC, December 2011.

13.4 Специальные темы квалификации инструмента

Есть несколько специальных тем, связанных с DO-330, которые заслуживают дополнительного пояснения.

13.4.1 Приказ FAA 8110.49

Глава 9 приказа 8110.49 Федерального управления гражданской авиации США содержит некоторые пояснения к DO-178B в части квалификации инструментов. Ожидается, что этот

раздел приказа будет удален или существенно изменен, поскольку DO-178C и DO-330 дают более полное руководство по квалификации инструментов, чем это делал DO-178B. Однако могут быть добавлены некоторые дополнительные указания, объясняющие переход от критериев квалификации инструментов по DO-178B к критериям DO-178C. Кроме того, может понадобиться руководство по тому, как использовать дополнения DO-331, DO-332 и DO-333 совместно с DO-330.

13.4.2 Детерминизм инструмента

В разделе 12.2 DO-178B содержалось следующее утверждение: «Квалифицироваться могут только детерминированные инструменты, то есть инструменты, которые при работе в одной и той же среде формируют один и тот же выход для одних и тех же входных данных» [5].

Приказ 8110.49 поясняет, что именно понимается под детерминированными инструментами. В разделе 9-6.d этого приказа объясняется, что детерминизм для инструментов часто трактуется слишком ограничительно. В приказе говорится:

Ограничительное толкование [детерминизма] состоит в том, что один и тот же кажущийся вход обязательно приводит в точности к одному и тому же выходу. Однако более точное толкование детерминизма для инструментов состоит в том, что установлена возможность определить корректность выхода инструмента. Если можно показать, что все возможные вариации выхода при некотором заданном входе являются корректными при любой надлежащей верификации этого выхода, то для целей квалификации инструмента такой инструмент следует считать детерминированным. Это превращает задачу в ограниченную. Такое толкование детерминизма должно применяться ко всем инструментам, выход которых может варьироваться вне контроля пользователя, но при этом эта вариативность не оказывает отрицательного влияния на предполагаемое использование, например на функциональность, выхода, и представлено обоснование корректности этого выхода. Однако такое толкование детерминизма не относится к инструментам, которые влияют на итоговый исполняемый образ, встраиваемый в бортовую систему. Формирование итогового исполняемого образа должно удовлетворять ограничительному толкованию детерминизма [6].[*]

DO-330 имеет тот же замысел, что и DO-178B и приказ 8110.49; однако слов *детерминированный* и *детерминизм* избегают, поскольку в программной инженерии они несут специальный смысл, который может быть чрезмерно ограничительным для инструментов, не являющихся летными. Вместо этого для пояснения ожиданий была сформулирована следующая позиция:

В ходе квалификации должно быть показано, что выход всех квалифицированных функций инструмента является корректным с использованием целей данного документа [DO-330]. Для инструмента, выход которого может в допустимых пределах варьироваться, должно

быть показано, что такая вариативность не оказывает отрицательного влияния на предполагаемое использование выходных данных и что корректность выходных данных может быть установлена. Однако для инструмента, выход которого является частью программного обеспечения и, следовательно, может внести ошибку, должно быть продемонстрировано, что квалифицированные функции инструмента выдают один и тот же результат для одних и тех же входных данных при работе в одной и той же среде [1].[*]

13.4.3 Дополнительные соображения по квалификации инструмента

В разделе 11 DO-330 содержится руководство по нескольким *дополнительным соображениям*, которые могут встретиться при разработке или использовании квалифицированных инструментов. Ниже эти дополнительные соображения кратко суммированы.

Дополнительное соображение 1: многофункциональные инструменты. Название говорит само за себя: многофункциональный инструмент представляет собой единый инструмент, выполняющий несколько функций. Эти функции могут находиться в одном исполняемом файле, в нескольких исполняемых файлах, что позволяет отключать отдельные функции, либо в иной конфигурации, допускающей выбор или отключение части функциональности инструмента [1]. В разделе 11.1 DO-330 поясняется, что многофункциональные инструменты должны быть описаны в планах и что для каждой функции необходимо определить TQL. Если присутствует смесь разных TQL, функции либо должны разрабатываться по наивысшему TQL, либо должны быть разделены так, чтобы функции с более низким TQL не влияли на функции с более высоким TQL, то есть чтобы обеспечивалась защита. Если какие-то функции использоваться не будут, их нужно отключить и иметь подтверждение того, что механизм отключения достаточно надежно защищает включенные функции от отключенных. Если неиспользуемые функции нельзя отключить, их нужно разрабатывать по соответствующему TQL, обычно по наивысшему TQL данного инструмента. Для инструментов, играющих роль и в процессах разработки, и в процессах верификации, необходимо учитывать независимость функций разработки инструмента и функций верификации, например они могут разрабатываться независимыми командами.

Дополнительное соображение 2: ранее квалифицированные инструменты. Раздел 11.2 DO-330 содержит рекомендации по повторному использованию инструментов, которые были квалифицированы ранее. В нем рассматриваются следующие сценарии повторного использования [1]:

- Инструмент и его эксплуатационная среда не изменились.
- Инструмент не изменился, но будет установлен в другой эксплуатационной среде.
- Инструмент изменился, но будет установлен в той же эксплуатационной среде.

- Инструмент изменился и будет установлен в другой эксплуатационной среде.

Возможность повторного использования существенно повышается, когда инструмент изначально спроектирован и упакован так, чтобы его можно было переиспользовать. Вопрос С.3 в DO-330 дает несколько рекомендаций о том, как разрабатывать и упаковывать инструмент для максимального повторного использования. Обычно повторное использование требует больше планирования, устойчивости и тестирования, но в дальнейшем может сэкономить значительные ресурсы.

Дополнительное соображение 3: квалификация готовых коммерческих инструментов. Раздел 11.3 DO-330 содержит руководство по успешной квалификации готового коммерческого инструмента. Готовые коммерческие инструменты уровня квалификации 5 квалифицировать относительно просто, поскольку понимание процесса разработки инструмента не требуется. Однако для инструментов уровней квалификации 1-4 требуется понимание самой разработки инструмента. По сути, готовые коммерческие инструменты все равно должны соответствовать целям DO-330.

Раздел 11.3 DO-330 объясняет, что обычно ожидается от разработчика инструмента и от пользователя инструмента при квалификации готового коммерческого инструмента. Важно помнить, что инструменты квалифицируются совместно с одобрением программного обеспечения по DO-178C, то есть квалификация инструмента не является самостоятельным одобрением. DO-330 пытается сделать квалификацию инструмента максимально автономной, но фактическая квалификация получается только в контексте сертификационного проекта. Для готовых коммерческих инструментов раздел 11.3 DO-330 поясняет, за какие цели обычно отвечает разработчик инструмента, а также дает предложения о том, как упаковать данные жизненного цикла, чтобы их можно было легче встроить в проект пользователя инструмента. В числе рекомендаций называются документы, подготавливаемые разработчиком: эксплуатационные требования к инструменту, план квалификации инструмента, указатель конфигурации инструмента и итоговый отчет по квалификации инструмента. Пользователь инструмента может оценить эти данные и либо использовать напрямую, либо ссылаться на них в собственных данных по квалификации инструмента. Замысел состоит в том, что разработчик инструмента предвосхищает потребности пользователя и заранее подготавливает данные квалификации инструмента так, чтобы этим потребностям соответствовать.

Дополнительное соображение 4: опыт эксплуатации инструмента. Раздел 11.4 DO-330 объясняет процесс на случай, если решено использовать опыт эксплуатации инструмента. Опыт эксплуатации может быть применим к инструменту со значительным опытом эксплуатации, но при отсутствии части данных жизненного цикла. Опыт эксплуатации так-

же может использоваться для повышения уровня квалификации инструмента, дополнения данных квалификации инструмента или увеличения уверенности в соответствии эксплуатационным требованиям к инструменту. Как и в случае опыта эксплуатации бортового программного обеспечения, см. главу 24, построить убедительное обоснование на основе опыта эксплуатации довольно сложно, и сертифицирующий орган, вероятно, воспримет его скептически.

Дополнительное соображение 5: альтернативные методы квалификации инструмента. Раздел 11.5 DO-330 объясняет, что заявитель может предложить альтернативные методы по сравнению с описанными в DO-330. Любая альтернатива должна быть описана в планах, тщательно обоснована, оценена с точки зрения влияния на процессы жизненного цикла и данные жизненного цикла результирующего программного обеспечения, а также согласована с сертифицирующим органом.

13.4.4 Подводные камни квалификации инструмента

За эти годы мне довелось оценивать десятки инструментов, и можно заметить ряд типичных подводных камней. Каждый проект отличается от других, и одни из этих сложностей встречаются чаще, чем другие. Однако они приведены здесь для повышения вашей осведомленности. Кто предупрежден, тот вооружен.

Подводный камень 1: отсутствуют инструкции для пользователя. Инструменты создаются для использования, как правило командой, которая их не разрабатывала. Однако инструкции для пользователя часто отсутствуют или недостаточны. Из-за этого неясно, будет ли инструмент применяться так, как задумано.

Подводный камень 2: не указана версия инструмента. Иногда существует несколько версий инструмента. Должно быть четко указано, какая именно версия используется. Обычно это документируется в Указателе конфигурации среды ЖЦ ПО или в Указателе конфигурации ПО. Также следует подтвердить, что команда действительно использует именно указанную версию.

Подводный камень 3: данные конфигурации не включены в пакет квалификации инструмента. Многие инструменты требуют некоторых конфигурационных данных. Например, инструмент, проверяющий соответствие стандарту MISRA C (стандарт Motor Industry Software Reliability Association для языка C), может быть активирован только для проверки части правил. Конкретные правила включаются или отключаются в конфигурационном файле. Такой файл должен находиться под управлением конфигурации и быть идентифицирован в квалификационных данных.

Подводный камень 4: потребность в квалификации инструмента оценена неточно. Иногда, анализируя данные команды, я обнаруживаю инструмент, который не был указан в

Плане сертификации ПО, потому что команда считала, что его не нужно квалифицировать; однако в действительности квалификация требуется. Подобные поздние открытия приводят к незапланированной работе. Как объяснялось в главе 5, чтобы избежать этого, рекомендуется перечислить в Плане сертификации ПО все инструменты и для каждого привести обоснование, почему его нужно или не нужно квалифицировать. Это позволяет проекту и сертифицирующему органу прийти к согласию на раннем этапе, а не в самом конце проекта.

Подводный камень 5: эксплуатационные требования к инструменту или требования к инструменту не проверяемы. Одни из худших требований, которые мне встречались, были именно требованиями к инструменту. Поскольку эксплуатационные требования к инструменту должны верифицироваться для всех уровней квалификации, а требования к инструменту верифицируются для уровней квалификации 1-4, требования должны быть проверяемыми. Эксплуатационные требования к инструменту и требования к инструменту должны обладать теми же качественными свойствами, которые обсуждались в главах 2 и 6.

Подводный камень 6: неполные квалификационные данные для готового коммерческого инструмента. Некоторые поставщики готовых коммерческих инструментов предлагают пакеты квалификационных данных. Некоторые из них действительно хороши, а некоторые – не очень. При вложении средств в пакет квалификации инструмента следует проявлять осторожность; разумно получить вместе с данными и какие-то гарантии.

Подводный камень 7: чрезмерная опора на инструменты без достаточного понимания их функциональности. Некоторые команды разработки программного обеспечения, особенно неопытные, полагаются на инструменты, не понимая по-настоящему, зачем они нужны, что именно делают и как вписываются в общий жизненный цикл ПО. Инструменты могут стать великолепным дополнением инженерного процесса. Однако их нужно понимать. Как любит говорить один мой знакомый: «Дурак с инструментом все равно остается дураком». Пользователи должны понимать, что именно инструмент делает для них.

Подводный камень 8: План сертификации ПО и Итоговый отчёт разработки ПО не объясняют подход к квалификации инструмента. Нередко в Плане сертификации ПО говорится, что инструмент нужно квалифицировать, но дальше этого дело не идет. Для инструментов уровней квалификации 1-4 План сертификации ПО должен кратко описывать использование инструмента и ссылаться на его План квалификации инструмента. Для уровня квалификации 5 подход к квалификации должен быть объяснен в Плане сертификации ПО или в Плане квалификации инструмента. Аналогично, Итоговый отчёт разработки ПО часто содержит недостаточно подробностей об инструменте и его квалификационных данных. Краткая сводка того, какая информация об инструменте должна включаться в План сертификации ПО и Итоговый отчёт разработки ПО, приведена в разделе 13.3.2, различие 5, этой

главы.

Подводный камень 9: команда тратит больше времени на разработку инструмента, чем на программное обеспечение, которое этот инструмент использует. Многим инженерам нравится разрабатывать инструменты; иногда они так увлекаются, что сосредотачиваются на инструменте настолько, что теряют из виду, ради чего он вообще нужен.

Подводный камень 10: роль инструмента или инструментов в общем жизненном цикле не объяснена в планах. Нередко описание процессов разработки, верификации, управления конфигурацией и обеспечения качества не упоминает инструменты, используемые в этих процессах. В плане инструменты перечислены в специальном разделе, но не объяснено, как они используются на этапах жизненного цикла, то есть там, где обсуждаются сами процессы. Например, инструмент управления требованиями может быть описан в разделе про инструменты, но не упомянут в разделах планов, посвященных сбору требований и их верификации. Из-за этого трудно убедиться, что инструменты действительно используются надлежащим образом в жизненном цикле.

Подводный камень 11: нечетко определен зачет квалификации. Когда инструменты заменяют, автоматизируют или устраняют цели DO-178C и/или приложений к нему, должно быть ясно, на какие цели они влияют. Это следует указывать в Плане сертификации ПО, а также в Плане разработки ПО, Плане верификации ПО и/или Плане УК ПО, где объясняется использование инструмента. Кроме того, для инструментов уровней квалификации 1-4 это может поясняться в Плане квалификации инструмента.

Подводный камень 12: эксплуатационная среда инструмента определена нечетко. Как уже упоминалось ранее в различиях 11 и 14 раздела 13.3.2, важно убедиться, что эксплуатационная среда, предполагаемая при квалификации, репрезентативно отражает среду, в которой инструмент используется на практике. Это часто упускается в Указателе конфигурации среды ЖЦ ПО и планах.

Подводный камень 13: код инструмента не архивируется. Поскольку инструменты являются частью среды разработки и верификации программного обеспечения, их следует архивировать вместе с другими данными жизненного цикла ПО. Для некоторых инструментов, требующих специального оборудования, возможно, придется архивировать и это оборудование.

13.4.5 DO-330 и приложения к DO-178C

Приложения по модельно-ориентированной разработке (DO-331), объектно-ориентированным технологиям (DO-332) и формальным методам (DO-333) применяют DO-330 так же, как и DO-178C.

По сути, квалификация инструмента одинакова вне зависимости от технологии, использующей этот инструмент. Однако, если сам инструмент реализован с применением модельно-ориентированной разработки, объектно-ориентированных технологий или формальных методов, то соответствующие технологические приложения, вероятно, придется применять к разработке и верификации самого инструмента.

13.4.6 Использование DO-330 в других доменах

DO-330 разрабатывался так, чтобы его можно было использовать в нескольких доменах. Если другой домен, например аэронавигационные базы данных или программируемая аппаратура, решит использовать DO-330, ему потребуется объяснить, как определять уровень квалификации инструмента для своего домена, как адаптировать терминологию программного обеспечения из DO-330 к своему домену и как уточнить любые специфичные для домена потребности. Приложение В DO-330 содержит пример того, как домен связи, навигации, наблюдения и организации воздушного движения реализовал документ квалификации инструмента DO-330. Другие домены могут использовать аналогичный подход или разработать собственный.

Ссылки

1. RTCA DO-330, *Software Tool Qualification Considerations* (Washington, DC: RTCA, Inc., December 2011).
2. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
3. RTCA/DO-254, *Design Assurance Guidance for Airborne Electronic Hardware* (Washington, DC: RTCA, Inc., April 2000).
4. RTCA/DO-200A, *Standards for Processing Aeronautical Data* (Washington, DC: RTCA, Inc., September 1998).
5. RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 1992).
6. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Change 1, September 2011).

*Совместно возглавлялось, в том числе, и автором этой книги.

*Различные уровни требований к инструменту следует тщательно согласовывать с сертифицирующим органом, поскольку по этой теме существуют расходящиеся мнения.

Скобки добавлены для ясности. Скобки добавлены для пояснения.

Глава 14. DO-331 и разработка и верификация на основе моделей

Сокращения

Сокращение	Расшифровка
EASA	European Aviation Safety Agency (Европейское агентство по авиационной безопасности)
EUROCAE	European Organization for Civil Aviation Equipment (Европейская организация по оборудованию для гражданской авиации)
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
TBY	Требование высокого уровня к ПО (high-level requirement)
THY	Требование низкого уровня к ПО (low-level requirement)
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
SC-205	Special Committee #205 (Специальный комитет N 205)
SCADE	Standard for the development of critical embedded display software (стандарт разработки критически важного встроенного ПО для отображения информации)
WG-71	Working Group #71 (Рабочая группа N 71)

14.1 Введение

Глоссарий DO-331 определяет *модель* следующим образом:

Абстрактное представление заданного набора аспектов системы, используемое для анализа, верификации, моделирования, генерации кода либо для любой комбинации перечисленного. Модель должна быть однозначной независимо от уровня своей абстракции.

Примечание 1. Если представление является диаграммой, допускающей неоднозначное толкование, оно не считается моделью.

Примечание 2. Под “заданным набором аспектов системы” могут пониматься все аспекты системы либо только их подмножество [1].

Модели могут входить в иерархию требований как системные требования, требования к ПО и/или проект ПО. Системные инженеры и инженеры по ПО годами использовали модели для графического представления функций управления. Однако применение квалифицированных инструментов моделирования, автоматически генерирующих код, а в некоторых случаях и автоматически генерирующих тестовые векторы, является для сообщества авиационного ПО сравнительно недавним сдвигом парадигмы. В Airbus A380 разработка на ос-

нове моделей широко использовалась для следующих систем: управление полетом, автопилот, предупреждение экипажа, индикация в кабине, управление топливом, шасси, торможение, руление, противообледенительная система и управление электрической нагрузкой [2]. Другие производители самолетов также моделируют требования для систем управления. На рисунках 14.1 и 14.2 приведен пример простой модели. Одна и та же модель показана в Simulink(R) (рисунок 14.1) и в стандарте разработки критически важного встроенного ПО для отображения информации (Standard for the Development of Critical Embedded Display Software, SCADE) (рисунок 14.2).

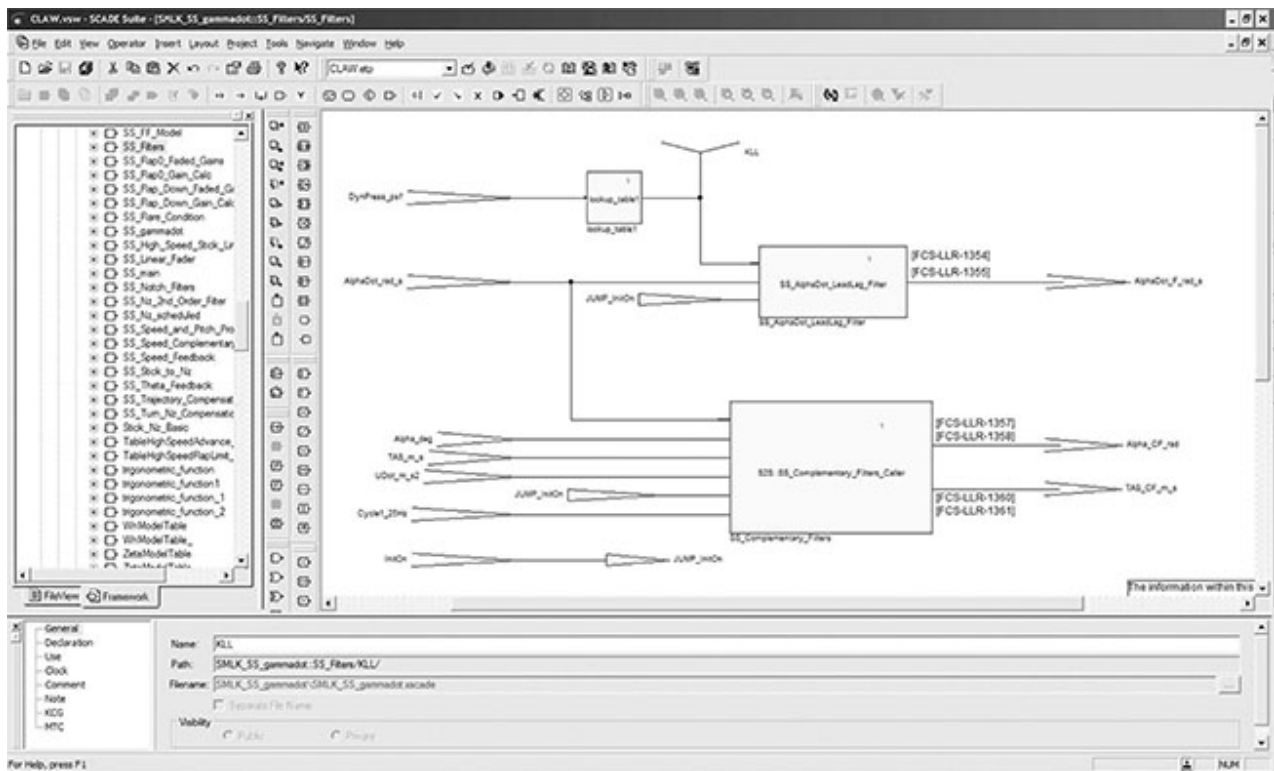


Рисунок 14.1 Пример модели в Simulink[R]. (Предоставлено Embraer, Сан-Паулу, Бразилия.)

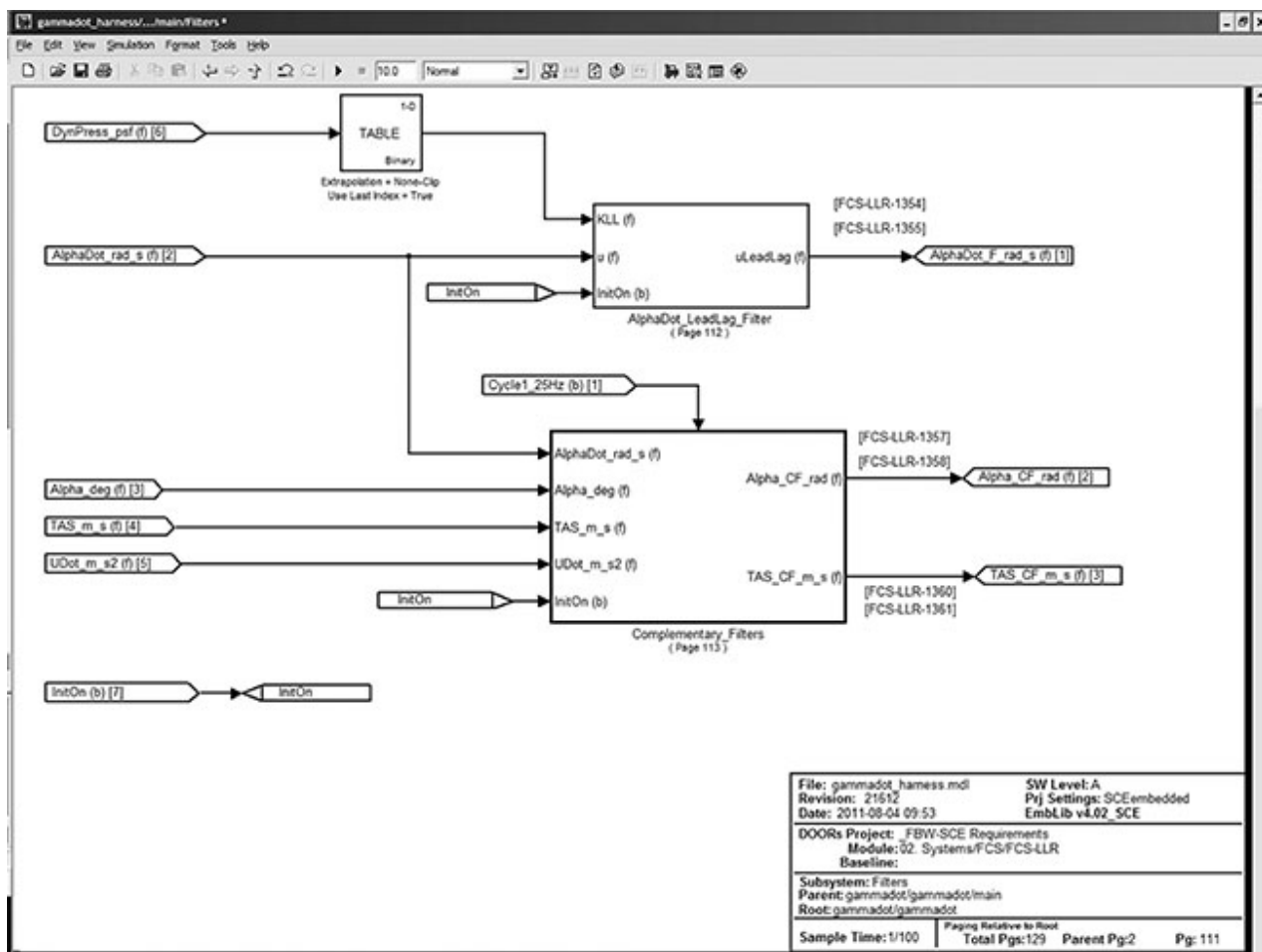


Рисунок 14.2 Пример модели в SCADE. (Предоставлено Embraer, Сан-Паулу, Бразилия.)

Такой сдвиг открывает возможность получить весьма значительные преимущества, но они не обходятся без определенных рисков и периода адаптации. В этой главе рассматриваются и достоинства, и недостатки разработки на основе моделей с точки зрения сертификации и безопасности. Кроме того, приводится краткий обзор DO-331, *Model-Based Development and Verification Supplement to DO-178C (KT-178C) and DO-278A* (дополнение к DO-178C и DO-278A по разработке и верификации на основе моделей).

14.2 Потенциальные преимущества разработки и верификации на основе моделей

Разработка и верификация на основе моделей дают ряд потенциальных преимуществ. Возможность реально получить эти преимущества зависит от деталей реализации, поэтому далеко не все из них обязательно будут достигнуты.

Потенциальное преимущество 1: переход от V-образного жизненного цикла к Y-образному жизненному циклу. Одним из главных мотиваторов разработки на основе моделей является возможность перейти от традиционного V-образного жизненного цикла к Y-образному жизненному циклу, что потенциально позволяет сократить сроки разработки, стоимость и, возможно, даже количество человеческих ошибок. Когда основной акцент переносится на системные требования и используется автоматизация, жизненный цикл может стать короче.

Некоторые оценки показывают сокращение примерно на 20% при использовании неквалифицированного генератора кода и вплоть до 50% при использовании квалифицированного генератора кода [3]. На рисунках 14.3 и 14.4 показан переход от V-образного к Y-образному жизненному циклу.



Рисунок 14.3 Традиционный V-образный жизненный цикл.



Рисунок 14.4 У-образный жизненный цикл при использовании разработки и верификации на основе моделей.

Существует несколько способов ввести модели в жизненный цикл продукта. Они могут создаваться на системном уровне, на уровне требований к ПО и/или на уровне проекта ПО. Большинство компаний стараются иметь модель платформенно-независимого уровня, которая абстрагирует модель от конкретной платформы и тем самым делает ее более переносимой. Детали, зависящие от платформы, могут быть представлены в модели более низкого уровня.

Потенциальное преимущество 2: больший фокус на требованиях. Как отмечалось в главе 6, ошибки в требованиях часто являются корневой причиной эксплуатационных проблем, связанных с ПО. Кроме того, чем позже обнаружена ошибка в требовании, тем дороже ее исправление. Когда модель представляет требования, она потенциально может давать более ясное представление о функциональности системы или ПО на этапе выявления требований, чем текстовые требования. Когда для реализации модели используется автоматизация, например квалифицированные генераторы автокода, команды разработки могут сосредоточиться на графическом представлении требований, а не на деталях реализации. При правильном подходе это способствует более высокому качеству требований и более раннему

обнаружению ошибок в требованиях.

Потенциальное преимущество 3: более ранняя верификация. При использовании квалифицированных инструментов преобразования модели в код фокус смещается с кода на модель, то есть меньше времени тратится на детали реализации кода и больше на саму модель. Использование моделирования и/или автоматических генераторов тестовых примеров способствует ранней верификации, позволяя раньше добиться зрелости требований и выявлять ошибки. На рисунках 14.3 и 14.4 показаны различия между традиционной разработкой и разработкой на основе моделей. Поскольку верификация часто потребляет около 50%-70% ресурсов ПО, более раннее обнаружение ошибок и сокращение объема ручной верификации могут благотворно сказаться на бюджете и графике проекта [4].

Потенциальное преимущество 4: уменьшение ненужной избыточности и несогласованностей. В традиционной разработке ПО используются три уровня требований: (1) системные требования, распределенные на ПО, (2) требования высокого уровня к ПО (ТВУ) и (3) требования низкого уровня к ПО (ТНУ). В зависимости от подхода к формированию требований между этими уровнями может возникать значительная избыточность, которая приводит к несогласованностям при внесении изменений. Использование квалифицированных инструментов моделирования потенциально позволяет уменьшить такую избыточность и число несогласованностей. Однако если имеется несколько уровней моделей, то может появиться и заметная доля собственной избыточности, а значит, и риск несогласованностей. Так что, как и в случае с другими преимуществами, все очень зависит от выбранного подхода.

Потенциальное преимущество 5: лучшая понятность требований. Некоторые инструменты создают модели, которые легче понимать, чем текстовые требования. Поэтому они могут способствовать большей точности, согласованности и полноте. Модели могут быть интуитивнее текстовых требований, благодаря чему ошибки и пробелы в запланированном поведении легче обнаруживаются [2]. Однако не все инструменты моделирования создают понятные модели, а некоторые из формируемых ими моделей, прямо скажем, весьма запутанны. Поэтому выбирать инструмент моделирования следует осторожно. Хороший стандарт моделирования также крайне важен, чтобы все разработчики корректно использовали нотации моделей, символы и инструменты.

Потенциальное преимущество 6: лучшее взаимодействие с заказчиками. Как обсуждалось в главе 6, участие заказчика критически важно для фиксации правильных требований. Благодаря более интуитивной природе моделей они могут быть полезны для получения обратной связи от заказчиков и системных инженеров. Это может усилить раннюю координацию между заказчиками, системными и программными командами, что повышает общее качество и точность требований.

Потенциальное преимущество 7: растущая поддержка со стороны инструментов. Возможности и качество инструментов разработки и верификации на основе моделей растут. Квалифицированных генераторов кода по-прежнему не так много, но они уже есть. Также доступны инструменты для верификации результатов работы генераторов кода. Когда полной уверенности в одном инструменте достичь нельзя, иногда возможно использовать два разных инструментальных тракта, чтобы усилить доверие к результату. Для некоторых более критичных систем, например систем управления полетом, такой подход может быть рекомендован даже тогда, когда доверие к инструментам в целом есть, поскольку он помогает снизить опасения относительно общей ошибки проекта.

Потенциальное преимущество 8: развитие поддержки формальных методов. Интересно, что, по-видимому, происходит сближение технологий моделирования, инструментальной поддержки и формальных методов, что может существенно увеличить преимущества разработки и верификации на основе моделей. Когда набор инструментов моделирования, включая генератор кода, формально верифицирован на корректность, на этот инструмент можно полагаться в части выполнения его предполагаемой функции. Без формальной верификации всегда остается неприятное ощущение, что инструмент моделирования или лежащий в его основе генератор кода может внести ошибку в каком-нибудь особенно неприятном граничном случае. Формальные методы, однако, могут обеспечить уверенность в комплекте инструментов преобразования модели в код, поскольку позволяют проверить все пространство входов и выходов. Подробнее о формальных методах см. в главе 16.

14.3 Потенциальные риски разработки и верификации на основе моделей

Как и у любой технологии, при использовании разработки и верификации на основе моделей есть ряд потенциальных сложностей. Ниже отмечены некоторые из них.

Потенциальный риск 1: потеря многократных рассмотрений требований. Когда модели реализуются с помощью автоматической генерации кода, часть традиционного критического анализа требований может быть утрачена, особенно если модель вводится на системном уровне. В традиционной разработке ПО выполняются рассмотрения системных требований, требований к ПО, проекта и кода. Кроме того, при анализе каждого уровня требований для получения нижележащего уровня выявляются и устраняются проблемы. Например, человек, реализующий код, может обнаружить несогласованность в требованиях. Матс Хеймдал называет это *попутной валидацией*. Он отмечает: “Опытные специалисты, проектирующие систему, разрабатывающие код или определяющие тестовые примеры, обеспечивают неформальную валидацию программной системы” [4]. Когда такие инженеры изучают данные, они замечают ошибки и могут обеспечить принятие корректирующих мер. Однако когда модель поднимается на более высокий уровень, а реализация передается инструментам,

эта *побочная валидация* сокращается либо исчезает совсем [4].

Потенциальный риск 2: расширение роли системной инженерии. В авиационной сфере, по крайней мере на нынешнем историческом этапе, инженеры по ПО обычно гораздо сильнее привержены процессной дисциплине, чем системные инженеры. DO-178B, а теперь и DO-178C, этому весьма способствовали. Системные инженеры, однако, лишь относительно недавно начали внедрять практики гарантии разработки. Поэтому, если модель находится под управлением системной инженерии, возникает беспокойство относительно качества модели с точки зрения ожиданий DO-178C и DO-331. Например, для системных инженеров может быть не вполне привычно обеспечивать соответствие стандартам, проводить документированные рассмотрения своих артефактов или тестировать все требования. Им также могут быть незнакомы такие понятия, как покрытие требований, тестирование на робастность, анализ покрытия модели и так далее. Чтобы преодолеть эту проблему, разумно усилить команду системной инженерии опытными инженерами по ПО. От такого устройства выигрывают и системные, и инженеры по ПО.

Потенциальный риск 3: сложности с трассируемостью. Даже если требования графически представлены в виде моделей, они все равно должны трассироваться вверх к требованиям более высокого уровня и вниз к требованиям более низкого уровня либо к коду. Трассируемость и вверх, и вниз должна быть двунаправленной. Некоторые инструменты моделирования плохо поддерживают трассируемость. Особенно сложной может оказаться связь между текстовыми требованиями и требованиями, представленными в виде модели.

Потенциальный риск 4: сложности с полнотой тестирования. В зависимости от подхода к моделированию и опыта команды тестировщиков может быть трудно гарантировать, что требования, представленные в форме модели, верифицированы полностью. Особенно сложными в этом отношении могут быть сложные модели. Нужны подробные рекомендации по тестированию, чтобы тесты, основанные на требованиях, действительно полностью проверяли модель. Кроме того, сложность модели также необходимо контролировать, чтобы обеспечить ее тестопригодность.

Потенциальный риск 5: спорность зачета моделирования. Одним из факторов, стимулирующих разработку и верификацию на основе моделей, является желание использовать инструменты моделирования для максимально раннего обнаружения ошибок. Однако когда компании пытаются получить формальный сертификационный зачет за такое моделирование, которое не выполняется на целевом вычислителе, это может становиться предметом споров. DO-331 пытается прояснить, что ожидается и что допускается в отношении моделирования, но до появления DO-331 это, как правило, обсуждалось с сертифицирующими органами в индивидуальном порядке для каждого случая. Даже при наличии DO-331 за-

прашиваемый сертификационный зачет за моделирование потребует тесной координации с сертифицирующими органами [*].

Потенциальный риск 6: модели часто смешивают что и как. Поскольку модели нередко содержат детали реализации, бывает трудно различить, что именно делает модель. По сути, проект и требования смешиваются. Как отмечалось в главе 6, это часто бывает проблемой и в традиционной разработке, но модели особенно предрасположены к такой дилемме. Чтобы предотвратить эту ситуацию, DO-331 требует наличия уровня требований над моделью. Предполагается, что такое руководство будет побуждать разработчиков явно определять, что именно модель должна делать. К сожалению, мне доводилось видеть довольно слабые требования над моделью. Например, однажды я видела модель на 98 страниц, над которой было одно-единственное требование. По сути, оно говорило: “У вас должна быть модель”. Проверять такое, прямо скажем, затруднительно.

Потенциальный риск 7: сложности разделения моделей спецификации и моделей проекта. Как будет вскоре показано, DO-331 дает рекомендации для моделей спецификации и моделей проекта. В руководстве сказано, что эти модели нельзя объединять.

На практике, однако, различать модели спецификации и модели проекта может быть непросто, особенно для уже существующих моделей. Большинство существующих моделей в той или иной степени совмещают что (требования) и как (проект). Кроме того, большинство техник моделирования поощряют включение проектных деталей, например уравнений и алгоритмов. Для большинства организаций разделение моделей спецификации и моделей проекта потребует весьма заметного сдвига в подходе.

Потенциальный риск 8: сложности валидации модели. Модели спецификации, особенно на системном уровне, необходимо валидировать на корректность и полноту. Без этого реализация модели имеет весьма ограниченную ценность. Зачастую процессы валидации модели развиты слабо. DO-331 пытается решить эту проблему, требуя анализа покрытия модели, который будет обсуждаться далее в этой главе, в дополнение к трассируемости к требованиям, расположенным над моделью.

Потенциальный риск 9: размывание границ между системными и программными ролями. В зависимости от того, на каком уровне вводится модель, системная команда может оказаться сильнее вовлеченной в детали нижнего уровня, чем это для нее привычно, а программная команда может сильнее, чем обычно, вовлекаться в детали верхнего уровня. Если все организовано правильно, более тесная связь между системными и программными инженерами может быть полезной. Но без должного внимания в процессе могут появиться либо ненужные перекрытия, либо большие пробелы.

Потенциальный риск 10: сложности совмещения традиционной и модельной разработки.

В большинстве проектов по разработке на основе моделей, помимо самих моделей, остаются и некоторые традиционные, то есть немодельные, требования и проект ПО. Точно так же даже при использовании генератора кода обычно требуется некоторый объем ручного кода. Сочетание традиционной и модельной разработки нужно тщательно определить, чтобы обеспечить согласованность и полноту. Кроме того, придется использовать и DO-178C, и DO-331: DO-178C для традиционной части и DO-331 для моделей. А это требует дополнительного обдумывания, координации и планирования. В планах должны быть четко определены разработка, верификация и интеграция как традиционных, так и модельных методов.

Потенциальный риск 11: непоследовательная интерпретация модели. Если нотация моделирования определена недостаточно четко и реализована недостаточно аккуратно, это может приводить к непоследовательной интерпретации модели. Подробные стандарты моделирования и соблюдение этих стандартов должны решать данную проблему.

Потенциальный риск 12: сопровождение модели. Без достаточной документации модели могут оказаться несопровождаемыми. Не так давно я участвовала в проекте по системе управления полетом. Инженер, создавший модель, ушел из компании. Другие ключевые инженеры тоже ушли. У команды, оставшейся завершать сертификационные работы, не было ни малейшего представления, что делает модель и почему она делает именно это. Существовала только сама модель. Не было требований более высокого уровня, допущений, обоснований или рациональных пояснений. Было неясно, почему были приняты те или иные решения и почему в модель были включены некоторые элементы.

Это был настоящий кошмар: валидировать корректность и полноту модели и верифицировать ее реализацию оказалось крайне трудно.

Мораль этой истории такова: сопровождаемые модели требуют тщательной документации. Так же как важно помещать комментарии в коде, см. главу 8, критически важно объяснять и обосновывать решения, принятые при моделировании. Раздел 14.4, различие 3, раскрывает эту идею подробнее.

Потенциальный риск 13: спорность автоматической генерации тестов. Некоторые инструменты моделирования не только автоматически генерируют код, но и автоматически генерируют тесты для модели. Это полезно для неформальной верификации и для повышения уверенности в модели. Однако когда проект пытается использовать автоматически сгенерированные тесты для сертификационного зачета, возникает целый ряд опасений. Если модель ошибочна, ошибка может остаться невыявленной. В традиционном тестировании ПО люди, разрабатывающие тесты, часто находят проблемы в требованиях и слабые места в системе, см. главу 9. Машинно-сгенерированные тесты могут оказаться не столь эффектив-

ными. Анализ покрытия модели может частично смягчить эти опасения, если он реализован правильно. Однако, помимо анализа покрытия модели, могут потребоваться и вручную созданные тесты, учитывающие интеграцию, требования безопасности, ключевую функциональность, сложные функции и робастность, чтобы дополнить автоматически сгенерированные тесты. При использовании автоматически сгенерированных тестов также необходимо обеспечить достаточную независимость: для более высоких уровней ПО генератор тестов и генератор кода должны быть разработаны независимо друг от друга.

Потенциальный риск 14: нестабильность инструментов. Поскольку разработка и верификация на основе моделей сильно опираются на инструменты, любая нестабильность инструментов может стать проблемой. Незрелые инструменты могут часто меняться, а это приводит к необходимости заново генерировать модели и/или выполнять их повторную верификацию. Недавно я слышала о проблеме, обнаруженной в инструменте моделирования, который использовался несколькими производителями самолетов и их поставщиками. Каждому пользователю инструмента пришлось оценивать влияние вновь обнаруженной проблемы на свой продукт. Волновой эффект от проблемы в инструменте в области критически важного по безопасности ПО может быть весьма значительным. Поэтому рекомендуется, если вы используете квалифицированные инструменты для поддержки разработки и верификации на основе моделей, удостовериться, что они достаточно зрелые и строгие для стоящей задачи. Независимая оценка инструмента и сопровождающих его данных может быть полезной, чтобы выявить его пригодность и зрелость.

Потенциальный риск 15: ограничения моделирования. Не всякая система подходит для моделирования. Разработка и верификация на основе моделей не обязательно будут правильным подходом для каждого проекта или каждой организации. Несмотря на некоторые заявления, это не серебряная пуля.

14.4 Обзор DO-331

DO-331 под названием *Model-Based Development and Verification Supplement to DO-178C and DO-278A* (дополнение к DO-178C и DO-278A по разработке и верификации на основе моделей) было самым сложным дополнением, разработанным Специальным комитетом N 205 (Special Committee #205, SC-205) RTCA и Рабочей группой N 71 (Working Group #71, WG-71) Европейской организации по оборудованию для гражданской авиации (EUROCAE). Это был также последний документ, одобренный комитетом, и он, возможно, наименее зрелый. Существует множество подходов к моделированию и множество мнений о том, как лучше внедрять модели в домене с критичностью по безопасности. Руководство DO-331 пытается обеспечить гибкость в реализации моделей, одновременно гарантируя, что ПО, генерируемое с использованием модели, выполняет свою предполагаемую функ-

цию и только ее. При правильном применении это дополнение помогает снять ряд опасений, упомянутых ранее в этой главе.

Как и другие дополнения, DO-331 использует структуру DO-178C и по мере необходимости изменяет, заменяет или добавляет цели, работы и руководящие указания к DO-178C. Определение модели в DO-331 приведено ранее в разделе 14.1. DO-331 признает, что модель может входить в ЖЦ ПО как системные требования, требования к ПО и/или проект ПО, а также что может существовать несколько уровней абстракции моделей. Независимо от того, на каком уровне модель входит в жизненный цикл ПО, над моделью и вне ее должны существовать требования, поясняющие детали и ограничения, чтобы можно было выполнять разработку и верификацию на основе моделей.

Раздел MB.1.6.2 документа DO-331 определяет два типа моделей: (1) *модель спецификации* и (2) *модель проекта*. Описание каждой приведено ниже [1]:[*]

Модель спецификации представляет требования высокого уровня, задающие абстрактное представление функциональных характеристик, характеристик производительности, интерфейсов или свойств безопасности компонентов ПО. Модель спецификации должна выражать эти характеристики недвусмысленно, чтобы поддерживать понимание функциональности ПО. Она должна содержать только те детали, которые способствуют такому пониманию, и не должна предписывать конкретную реализацию или архитектуру ПО, кроме исключительных случаев оправданных проектными ограничениями. Модели спецификации не определяют такие детали проекта ПО, как внутренние структуры данных, внутренние потоки данных или внутренние потоки управления. Следовательно, модель спецификации может выражать требования высокого уровня, но не требования низкого уровня и не архитектуру ПО.

Модель проекта предписывает внутренние структуры данных компонента ПО, потоки данных и/или потоки управления. Модель проекта включает требования низкого уровня и/или архитектуру. В частности, если модель выражает данные проекта ПО, независимо от прочего содержимого ее следует классифицировать как модель проекта. Это относится и к моделям, используемым для получения кода.

Следует отметить две важные вещи относительно этих типов моделей. Во-первых, модель не может одновременно классифицироваться и как модель спецификации, и как модель проекта. Во-вторых, поскольку DO-331 требует, чтобы над моделью существовали требования, при использовании моделей всегда будет как минимум два уровня требований [1].

DO-331 сохраняет большую часть руководящих указаний DO-178C, но добавляет информацию, специфичную для моделей. В общем случае большая часть руководства DO-178C, относящегося к требованиям высокого уровня к ПО (ТВУ), применяется к моделям специ-

фикации, а руководство по проекту ПО относится к моделям проекта. Наиболее значимые различия между DO-331 и DO-178C сведены ниже. Именно в этих областях DO-331 изменяет или уточняет, то есть дополняет, DO-178C применительно к разработке и верификации на основе моделей.

Различие 1: планирование моделей. На этапе планирования в планах следует объяснить использование моделирования и то, как оно встраивается в ЖЦ ПО. Планы должны объяснять, какие данные ЖЦ ПО представляет каждая модель, какие стандарты моделирования будут использоваться и какой подход к верификации предполагается. Если моделирование будет использоваться для сертификационного зачета, планы должны явно описывать подход и какой именно зачет запрашивается. На этапе планирования также следует определить среду моделирования, включая методы, инструменты, процедуры и операционную среду, если моделирование используется для формальной поддержки верификации. В зависимости от того, как используется симулятор, может потребоваться его квалификация; в планы следует включить обоснование того, почему квалификация нужна или не нужна. Глава 13 объясняет критерии и подход к квалификации инструмента.

Если помимо моделей используются и *традиционные* текстовые требования и элементы проекта, это должно быть объяснено в планах. В такой ситуации планы должны пояснять, как будут удовлетворены и цели DO-178C, и цели DO-331. Существует несколько способов этого добиться. Один из них состоит в том, чтобы включить в план сертификации ПО и цели DO-178C, и цели DO-331, возможно в виде приложения, чтобы показать, как будут удовлетворяться оба набора целей.

Различие 2: стандарты моделей. Для каждого используемого типа модели должны существовать стандарты, объясняющие техники моделирования, ограничения, указания и так далее. Раздел MB.11.23 документа DO-331 объясняет, что для каждого типа модели должен существовать стандарт модели и что каждый такой стандарт должен как минимум включать следующие виды информации [1]:

- Пояснение и обоснование используемых методов и инструментов. Указание языка моделирования, который будет использоваться, а также любого синтаксиса языка, семантики, возможностей и ограничений.
- Руководящие указания по стилю и ограничения сложности, позволяющие сделать подход к моделированию и его реализацию недвусмысленными, детерминированными и соответствующими целям DO-331. Ограничения сложности особенно важны. Обычно нужно ограничивать глубину модели, то есть уровни вложенности, число архитектурных слоев и число элементов на диаграмму.
- Ограничения, обеспечивающие правильное использование инструментов моделирова-

ния и поддерживающих библиотек.

- Методы идентификации требований, установления двунаправленной трассируемости между уровнями требований, идентификации производных требований, документирования обоснования для всех производных требований и идентификации любого элемента модели, который не является требованием или проектом ПО, например комментариев.

Стандарты моделей служат ориентиром для разработчиков, позволяя им создавать качественные и соответствующие требованиям модели. Полезны четкие указания и реалистичные примеры.

Различие 3: вспомогательные элементы модели. DO-331 объясняет, что элементы модели, не вносящие вклада в реализацию требований или проекта, должны быть четко идентифицированы. Для решения этой задачи DO-331 добавляет три цели. Все три цели находятся в таблице MB.A-2 приложения MB документа DO-331 и перечислены ниже [1]:

- *Цель MB8:* «Идентифицированы элементы модели спецификации, не вносящие вклада в реализацию какого-либо требования высокого уровня».
- *Цель MB9:* «Идентифицированы элементы модели проекта, не вносящие вклада в реализацию какой-либо архитектуры ПО».
- *Цель MB10:* «Идентифицированы элементы модели проекта, не вносящие вклада в реализацию какого-либо требования низкого уровня».

Различие 4: библиотеки элементов модели. В большинстве инструментов моделирования библиотеки используются очень широко. Например, символы, применяемые для графического представления модели, берутся из библиотеки символов. Для каждого элемента библиотеки, который может использоваться в модели, должна быть обеспечена надлежащая гарантия на соответствующем уровне ПО согласно DO-178C. По сути, библиотечным элементам требуются планы, стандарты разработки, требования, проект, код, примеры и процедуры верификации и так далее, как и любому другому критически важному по безопасности ПО. Если в библиотеке есть элементы, не обладающие нужным уровнем гарантии, их не следует использовать. Предпочтительно вообще удалить их из библиотеки, чтобы избежать непреднамеренного использования. Однако если это невозможно, должны существовать явные стандарты, запрещающие использование элементов без требуемой гарантии, а также рассматривания, подтверждающие соблюдение этих стандартов. Кроме того, стандарты моделирования должны давать указания по правильному использованию библиотечных элементов.

Различие 5: анализ покрытия модели для моделей проекта. Глоссарий DO-331 определяет анализ покрытия модели следующим образом:

Анализ, определяющий, какие требования, выраженные моделью проекта, не были задействованы верификацией, основанной на требованиях, из которых эта модель проекта была разработана. Цель такого анализа состоит в поддержке обнаружения непредусмотренной функции в модели проекта в условиях, когда покрытие требований, из которых была разработана модель, достигнуто примерами верификации [1].

Анализ покрытия модели не тождествен анализу структурного покрытия. Раздел MB.6.7 документа DO-331 объясняет рекомендуемые мероприятия и критерии анализа покрытия модели, а также мероприятия по устранению замечаний, если выявлены проблемы покрытия. Любопытно, что на раздел MB.6.7 в таблице MB.A-4 приложения MB к DO-331 есть ссылка как на мероприятие, а не как на цель, то есть ссылка MB.6.7 находится в колонке *Мероприятие* (Activity), а не в колонке *Цель* (Objective). Цель 4 гласит: «Достигнуто покрытие требований низкого уровня тестированием». Анализ покрытия модели нужен только для моделей проекта. Его включение в DO-331 было в некоторой степени спорным, поэтому он был включен как мероприятие, а не как цель. Несмотря на то что в таблице MB.A-4 документа DO-331 он указан только как мероприятие, а не как цель, большинство сертифицирующих органов будут ожидать, что он выполнен. Они, вероятно, допустят использование альтернативных подходов, но им все равно потребуются свидетельства того, что модель проекта была верифицирована полностью с тем же уровнем строгости, который обеспечивает анализ покрытия модели.

Различие 6: моделирование. Глоссарий DO-331 определяет *моделирование* и *симулятор модели* следующим образом [1]:

Моделирование: мероприятие по проигрыванию поведения модели с использованием симулятора модели.

Симулятор модели: устройство, компьютерная программа или система, позволяющие исполнять модель для демонстрации ее поведения в поддержку верификации и/или валидации.

Примечание: симулятор модели может как исполнять, так и не исполнять код, который репрезентативен по отношению к целевому коду.

Раздел MB.6.8 документа DO-331 содержит специальные указания по моделированию. Моделирование может использоваться для удовлетворения некоторых целей верификации DO-331. В таблице 14.1 суммированы цели DO-331, которые могут или не могут удовлетворяться с использованием моделирования. Поскольку модель может представлять требования высокого уровня к ПО (ТВУ), требования низкого уровня к ПО (ТНУ) или архитектуру ПО, цели из таблицы 14.1 применяются к соответствующему представлению модели.

Таблица 14.1 Сводка целей, связанных с моделированием

Цель DO-331	Ссылка	Действие по верификации	Допускается ли моделирование
Таблица MB.A-3, цель 1	DO-331 MB.6.3.1.a	Соответствие системным требованиям для требований высокого уровня	Да
Таблица MB.A-3, цель 2	DO-331 MB.6.3.1.b	Точность и согласованность требований высокого уровня	Да
Таблица MB.A-3, цель 3	DO-331 MB.6.3.1.c	Совместимость требований высокого уровня с целевым вычислителем	Нет
Таблица MB.A-3, цель 4	DO-331 MB.6.3.1.d	Проверяемость требований высокого уровня	Да
Таблица MB.A-3, цель 5	DO-331 MB.6.3.1.e	Соответствие требований высокого уровня стандартам	Нет
Таблица MB.A-3, цель 6	DO-331 MB.6.3.1.f	Двунаправленная трассируемость между требованиями высокого уровня и системными требованиями	Нет
Таблица MB.A-3, цель 7	DO-331 MB.6.3.1.g	Алгоритмические аспекты требований высокого уровня	Да

Цель DO-331	Ссылка	Действие по верификации	Допускается ли моделирование
Таблица MB.A-4, цель 1	DO-331 MB.6.3.2.a	Соответствие требований высокого уровня требованиям низкого уровня к ПО	Да
Таблица MB.A-4, цель 2	DO-331 MB.6.3.2.b	Точность и согласованность требований низкого уровня	Да
Таблица MB.A-4, цели 3 и 10	DO-331 MB.6.3.2.c, MB.6.3.3.c	Совместимость с целевым вычислителем, требованиями низкого уровня и архитектурой ПО	Нет
Таблица MB.A-4, цели 4 и 11	DO-331 MB.6.3.2.d, MB.6.3.3.d	Проверяемость требований низкого уровня и архитектуры ПО	Да
Таблица MB.A-4, цели 5 и 12	DO-331 MB.6.3.2.e, MB.6.3.3.e	Соответствие требований низкого уровня стандартам	Нет
Таблица MB.A-4, цель 6	DO-331 MB.6.3.2.f	Двунаправленная трассируемость между требованиями низкого и высокого уровня	Нет
Таблица MB.A-4, цель 7	DO-331 MB.6.3.2.g	Алгоритмические аспекты требований низкого уровня	Да

Цель DO-331	Ссылка	Действие по верификации	Допускается ли моделирование
Таблица MB.A-4, цель 8	DO-331 MB.6.3.3.a	Совместимость архитектуры ПО с требованиями высокого уровня	Да
Таблица MB.A-4, цель 9	DO-331 MB.6.3.3.b	Согласованность архитектуры ПО	Да
Таблица MB.A-4, цель 13	DO-331 MB.6.3.3.f	Подтверждена целостность обособления	Нет
Таблица MB.A-6, цель 1	DO-178C 6.4.a	Исполняемый объектный код соответствует требованиям высокого уровня	Частично
Таблица MB.A-6, цель 2	DO-178C 6.4.b	Исполняемый объектный код устойчив по отношению к требованиям высокого уровня	Частично
Таблица MB.A-6, цель 3	DO-178C 6.4.c	Исполняемый объектный код соответствует требованиям низкого уровня	Нет
Таблица MB.A-6, цель 4	DO-178C 6.4.d	Исполняемый объектный код устойчив по отношению к требованиям низкого уровня	Нет

Цель DO-331	Ссылка	Действие по верификации	Допускается ли моделирование
Таблица MB.A-6, цель 5	DO-178C 6.4.e	Исполняемый объектный код совместим с целевым вычислителем	Нет
Таблица MB.A-7, цель 1	DO-178C 6.4.5.b	Тестовые процедуры корректны	Нет
Таблица MB.A-7, цель 2	DO-178C 6.4.5.c	Результаты тестирования корректны, а расхождения объяснены	Нет
Таблица MB.A-7, цель 3	DO-178C 6.4.4.a	Достигнуто покрытие требований высокого уровня тестированием	Частично
Таблица MB.A-7, цель 4	DO-178C 6.4.4.b	Достигнуто покрытие требований низкого уровня тестированием	Нет
Таблица MB.A-7, цели 5-7	DO-178C 6.4.4.c	Достигнуто покрытие структуры ПО до требуемого критерия	Частично
Таблица MB.A-7, цель 8	DO-178C 6.4.4.d	Достигнуто покрытие структуры ПО, а также покрытие по связи данных и управления	Частично

Цель DO-331	Ссылка	Действие по верификации	Допускается ли моделирование
Таблица MB.A-7, цель 9	DO-178C 6.4.4.c	Выполнена верификация дополнительного кода, который нельзя трассировать к исходному коду	Нет

Источник: RTCA DO-331, *Model-Based Development and Verification Supplement to DO-178C and DO-278A*, RTCA, Inc., Washington, DC, December 2011.

Примечание: даже при частично допускаемом моделировании часть тестирования по-прежнему требуется выполнять на целевом вычислителе.

Если примеры и процедуры моделирования используются для формального зачета при верификации, примеры и процедуры моделирования должны быть верифицированы на корректность, а результаты моделирования должны быть рассмотрены, причем любые расхождения в результатах должны быть объяснены. DO-331 добавляет новые цели в таблицы MB.A-3 приложения MB, цели MB8-MB10, MB.A-4, цели MB14-MB16, и MB.A-7, цели MB10-MB12, чтобы охватить верификацию примеров моделирования, процедур и результатов. Эти три цели одинаковы, но повторяются в трех таблицах приложения, поскольку относятся к разным этапам ЖЦ ПО [1]:

- «Примеры моделирования корректны». (Таблица MB.A-3, цель MB8; таблица MB.A-4, цель MB14; и таблица MB.A-7, цель MB10.)
- «Процедуры моделирования корректны». (Таблица MB.A-3, цель MB9; таблица MB.A-4, цель MB15; и таблица MB.A-7, цель MB11.)
- «Результаты моделирования корректны, а расхождения объяснены». (Таблица MB.A-3, цель MB10; таблица MB.A-4, цель MB16; и таблица MB.A-7, цель MB12.)

14.5 Признание DO-331 сертифицирующими органами

До публикации DO-331 международные сертифицирующие органы использовали проектно-специфические документы по проблемным вопросам (issue papers) или их эквиваленты для выявления вопросов разработки и верификации на основе моделей, которые необходимо было решить.[*] Предполагается, что при использовании DO-331 как средства подтверждения соответствия такие документы по проблемным вопросам или их эквиваленты больше не понадобятся. По мере накопления опыта применения DO-331 в будущем могут появиться дополнительные указания сертифицирующих органов, уточняющие вопросы разработки

и верификации на основе моделей.

Из всех дополнений к DO-178C именно DO-331, вероятно, будет сложнее всего применять, особенно при попытке распространить его указания на уже существующие модели. Некоторые конкретные сложности таковы:

- Разработка полного набора требований над моделью, особенно когда модель находится вне ЖЦ ПО.
- Уточнение области применения DO-331 в системном домене.
- Разделение модели спецификации и модели проекта.
- Выполнение анализа покрытия модели.
- Выполнение двунаправленной трассируемости между элементами модели и верхними и нижними уровнями.
- Интеграция новых указаний в существующие процессы.
- Использование моделирования для получения сертификационного зачета. Автоматическая генерация тестов по моделям.

Ссылки

1. RTCA DO-331, *Model-Based Development and Verification Supplement to DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
2. S. P. Miller, Proving the shalls: Requirements, proofs, and model-based development, Federal Aviation Administration 2005 Software & Complex Electronic Hardware Conference (Norfolk, VA, 2005).
3. B. Dion, Efficient development of airborne software using model-based development, 2004 *FAA Software Tools Forum* (Daytona, FL, May 2004).
4. M. Heimdahl, Safety and software intensive systems: Challenges old and new, Future of Software Engineering Conference (Minneapolis, MN, 2007).

*По мере накопления опыта применения DO-331 это область, в которой сертифицирующие органы, вероятно, выработают дополнительные указания.

*Курсив добавлен для ясности.

*Например, сертификационный меморандум Европейского агентства по авиационной безопасности (EASA) CM-SWCEH-002 («Software Aspects of Certification») от 11 августа 2011 года включает раздел по разработке на основе моделей (раздел 23). На этот меморандум ссылаются элементы сертификационного рассмотрения (certification review items) Европейского агентства по авиационной безопасности, которые являются эквивалентом документов

по проблемным вопросам Федерального управления гражданской авиации США.

Глава 15. DO-332 и объектно-ориентированная технология и связанные с ней методы

Сокращения

Сокращение	Расшифровка
AVSI	Aerospace Vehicle Systems Institute (Институт систем аэрокосмических аппаратов)
CAST	Certification Authorities Software Team (Группа сертифицирующих органов по ПО)
CRI	Certification review item (элемент сертификационного рассмотрения)
DC/CC	Data coupling and control coupling (межкомпонентная связность по данным и по управлению)
EASA	European Aviation Safety Agency (Европейское агентство по авиационной безопасности)
FAA	Federal Aviation Administration (Федеральное управление гражданской авиации США)
FAQ	Frequently asked questions (часто задаваемые вопросы)
IEEE	Institute of Electrical and Electronics Engineers (Институт инженеров электротехники и электроники)
LSP	Liskov Substitution Principle (принцип подстановки Лисков)
NASA	National Aeronautics and Space Administration (Национальное управление по аэронавтике и исследованию космического пространства)
OOT	Object-oriented technology (объектно-ориентированная технология)
OOTiA	Object-oriented technology in aviation (объектно-ориентированная технология в авиации)
OOT&RT	Object-oriented technology and related techniques (объектно-ориентированная технология и связанные с ней методы)

15.1 Введение в объектно-ориентированную технологию

Объектно-ориентированная технология (object-oriented technology, OOT) существует с конца 1960-х годов, когда появился язык программирования Simula [1]. Объектно-

ориентированная технология представляет собой парадигму разработки ПО для анализа, проектирования, моделирования и программирования, в центре которой находятся *объекты*. Институт инженеров электротехники и электроники (Institute of Electrical and Electronics Engineers, IEEE) определяет объектно-ориентированную технологию как «технику разработки ПО, при которой система или компонент выражаются в терминах объектов и связей между этими объектами» [2]. Объект подобен *черному ящику* на уровне ПО; каждый объект способен получать сообщения, обрабатывать данные и отправлять сообщения другим объектам. Объект содержит как код (методы), так и данные (структуры). Пользователю не требуется понимать внутренние детали объекта, чтобы использовать его, отсюда и сравнение с черным ящиком. Объект может моделировать сущности реального мира, например датчик или аппаратный контроллер, в виде отдельных компонентов ПО с определенным поведением. Одним из основных понятий объектно-ориентированной технологии является *класс*. Класс определяет атрибуты, методы, отношения и семантику, имеющие общую структуру и поведение, характерные для сущности реального мира.

Раздел ОО.1.6.1.1 документа DO-332 дает следующее описание объектов и классов [3]:

Отличительной особенностью объектно-ориентированной технологии является использование классов для определения объектов вместе с возможностью создавать новые классы посредством наследования. В процедурном программировании поведение программы определяется функциями, а состояние выполняющейся программы определяется содержимым переменных данных. В объектно-ориентированном программировании функции и данные, тесно связанные между собой, объединяются, образуя согласованную абстракцию, известную как класс...

Класс представляет собой шаблон, на основе которого можно создать несколько конкретных реализаций. Эти конкретные реализации известны как объекты...

Подпрограммы, определенные для класса, называются его методами, или функциями-членами. Методы, которые работают с данными, содержащимися в экземпляре объекта, или используют их, называются методами экземпляра. В отличие от них методы класса связаны только с классом и не требуют связанного экземпляра для вызова.

Во всем документе DO-332 термин *подпрограмма* используется как общее обозначение для всех форм методов (методов экземпляра и методов класса), функций или процедур. DO-332 также объясняет:

Переменные данных, связанные с классом, называются атрибутами, полями или членами данных. Атрибут может быть классифицирован как атрибут экземпляра, и тогда для каждого объекта существует отдельная копия атрибута, либо как атрибут класса, и тогда существует одна общая копия атрибута для всех объектов класса [3].

15.2 Использование объектно-ориентированной технологии в авиации

Объектно-ориентированная технология широко применяется при разработке ПО, некритичного по безопасности, например в веб-приложениях и настольных приложениях, и преподается почти исключительно в университетах. Она также использовалась в критичных по безопасности медицинских и автомобильных системах. Объектно-ориентированная технология привлекательна для авиационной отрасли по нескольким причинам, включая сильную инструментальную поддержку, наличие программистов, предполагаемое снижение затрат и потенциал повторного использования.

Однако на сегодняшний день объектно-ориентированная технология не получила широкого распространения в авиации. Федеральное управление гражданской авиации США (Federal Aviation Administration, FAA) и авиационная отрасль более 10 лет исследовали и изучали применение объектно-ориентированной технологии в критичных по безопасности системах и вырабатывали руководящие указания для ее безопасной реализации. Существует несколько технических сложностей, связанных с объектно-ориентированной технологией, в основном относящихся к языкам программирования, которые задержали ее широкое использование в системах реального времени и в авиации. Для проектов, использующих объектно-ориентированную технологию, выпускались документы по проблемным вопросам Федерального управления гражданской авиации США и элементы сертификационного рассмотрения (Certification review item, CRI) Европейского агентства по авиационной безопасности (European Aviation Safety Agency, EASA), чтобы гарантировать рассмотрение этих вопросов.

15.3 Справочник по объектно-ориентированной технологии в авиации

В 1999 году Федеральное управление гражданской авиации США и отрасль начали активно изучать применение объектно-ориентированной технологии в авиации. Несколько проектов рассматривали возможность использования этого подхода, однако исследований, позволяющих оценить его пригодность для критичных по безопасности приложений и приложений реального времени, почти не существовало. Команда Института систем аэрокосмических аппаратов (Aerospace Vehicle Systems Institute, AVSI)[*], которую поддерживали Boeing, Honeywell, Goodrich и Rockwell Collins, совместно работала над проектом под названием «Вопросы сертификации встроенного объектно-ориентированного ПО», целью которого было снизить риск, с которым сталкиваются отдельные проекты при сертификации бортовых систем с объектно-ориентированным ПО. В рамках проекта Института систем аэрокосмических аппаратов был предложен ряд руководящих указаний по созданию объектно-ориентированного ПО, соответствующего DO-178B [4]. Работа Института систем аэрокосмических аппаратов стала входным материалом для совместной инициати-

вы Федерального управления гражданской авиации США и Национального управления по аэронавтике и исследованию космического пространства (National Aeronautics and Space Administration, NASA) под названием «Объектно-ориентированная технология в авиации» (Object-oriented technology in aviation, OOTiA). Исследовательский центр Langley Национального управления по аэронавтике и исследованию космического пространства и Федеральное управление гражданской авиации США провели два семинара для сбора мнений отрасли и совместной разработки четырехтомного документа под названием *Handbook for Object-Oriented Technology in Aviation*, который был завершен в октябре 2004 года [5].

Проблемы, выявленные в *OOTiA Handbook*, стали основой для оценки объектно-ориентированной технологии в авиационных проектах. От заявителей и разработчиков ПО требовалось обеспечить, чтобы их подход к объектно-ориентированной технологии учитывал проблемы, выявленные во втором томе *OOTiA Handbook*. В третьем томе *OOTiA Handbook* предлагались некоторые способы решения проблем, поднятых в следующих 10 областях: (1) одиночное наследование и динамическая диспетчеризация, (2) множественное наследование, (3) шаблоны, (4) встраивание, (5) преобразование типов, (6) перегрузка и разрешение методов, (7) мертвый и отключенный код, а также повторное использование, (8) объектно-ориентированные инструменты, (9) трассируемость и (10) структурное покрытие.

15.4 Исследования объектно-ориентированной технологии и структурного покрытия, финансируемые Федеральным управлением гражданской авиации США

В период 2002-2007 годов Федеральное управление гражданской авиации США финансировало трехэтапную исследовательскую программу, в рамках которой рассматривались вопросы структурного покрытия, способные возникать при использовании объектно-ориентированной технологии.[*] На первом этапе был выполнен обзор проблем, по итогам которого появился отчет под названием *Issues concerning the structural coverage of object-oriented software* [6].

На втором этапе исследовались проблемы и предлагались критерии приемки для подтверждения межкомпонентной связности по данным и по управлению (data coupling and control coupling, DC/CC) в рамках объектно-ориентированной технологии в коммерческой авиации. Как отмечалось в главе 9, цель подтверждения межкомпонентной связности по данным и по управлению состоит в том, чтобы обеспечить объективную оценку, то есть измерение, полноты тестов, основанных на требованиях, для интегрированных компонентов. По итогам второго этапа появился отчет под названием *Object-oriented technology verification phase 2 report-Data coupling and control coupling* [7]. Этот отчет содержал рекомендации по покрытию межкомпонентных зависимостей как измеримому критерию достаточности для

выполнения цели 8 таблицы A-7 DO-178B, а теперь и DO-178C (KT-178C). В дополнение к отчету в рамках второго этапа был подготовлен справочник под названием *Object-oriented technology verification phase 2 handbook-Data coupling and control coupling* [8]. Этот справочник содержит указания по проблемам и критериям приемки для подтверждения межкомпонентной связности по данным и по управлению в рамках объектно-ориентированной технологии в коммерческой авиации.

На третьем этапе исследовались проблемы и критерии приемки, связанные с использованием анализа структурного покрытия на уровне исходного кода по сравнению с уровнем объектного кода или исполняемого объектного кода в рамках объектно-ориентированной технологии, чтобы выполнить цели 5-8 таблицы A-7 DO-178B, а теперь и DO-178C. Как и второй этап, третий этап завершился выпуском отчета и справочника:

- *Object-oriented technology verification phase 3 report-Structural coverage at the source-code and object-code levels* [9]
- *Object-oriented technology verification phase 3 handbook-Structural coverage at the source-code and object-code levels* [10]

15.5 Обзор DO-332

Документ DO-332 называется *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A* (Дополнение по объектно-ориентированной технологии и связанным с ней методам к DO-178C и DO-278A). Он содержит указания по использованию объектно-ориентированной технологии и методов, тесно связанных с ней. DO-332 также называют *дополнением по объектно-ориентированной технологии и связанным с ней методам*. В качестве входных материалов для DO-332 использовались *OOTiA Handbook*, исследовательские отчеты Федерального управления гражданской авиации США, документы по проблемным вопросам Федерального управления гражданской авиации США, элементы сертификационного рассмотрения Европейского агентства по авиационной безопасности и позиционные документы группы сертифицирующих органов по ПО. Дополнение изменяет и расширяет цели DO-178C, мероприятия, пояснительный текст и данные ЖЦ ПО в тех случаях, когда в ЖЦ ПО используются объектно-ориентированная технология и связанные с ней методы. Поскольку терминология объектно-ориентированной технологии и общее понимание этого подхода в авиационной отрасли различаются, дополнение дает обзор объектно-ориентированной технологии и связанных с ней методов. К связанным методам относятся параметрический полиморфизм, перегрузка, преобразование типов, управление исключениями, динамическое управление памятью и виртуализация. Большинство, но не все объектно-ориентированные языки используют эти методы. Кроме того, некоторые из этих методов применимы и за пределами объектно-ориентированной технологии. Ниже

кратко изложены основные различия между DO-332 и DO-178C.

15.5.1 Планирование

DO-332 добавляет в процесс планирования DO-178C три мероприятия. Во-первых, если используется виртуализация, это должно быть объяснено в планах. Во-вторых, если компоненты используются повторно, а это одна из целей объектно-ориентированной технологии, такое повторное использование должно быть описано в планах, включая «поддержание согласованности типов, сопоставление требований и стратегию управления исключениями между компонентами и использующей системой» [3]. В-третьих, планы и стандарты должны объяснять, каким образом будут устраняться уязвимости из приложения OO.D документа DO-332, обсуждаемые в разделе 15.5.4.

15.5.2 Разработка

DO-332 добавляет некоторые дополнительные указания по разработке, специфичные для объектно-ориентированной технологии, сверх того, что уже включено в DO-178C. В частности, раздел OO.5 документа DO-332 добавляет указания по следующим темам: иерархия классов, согласованность типов, управление памятью, управление исключениями и отключенная функциональность при повторном использовании [3].

Раздел OO.5.5 документа DO-332 также уточняет вопросы трассируемости для объектно-ориентированной технологии. Поскольку объектно-ориентированный проект ПО реализуется с использованием методов, трассируемость ведется от требований к методам и атрибутам, реализующим эти требования. Классы являются артефактом архитектуры, предназначенным для организации требований. Из-за наличия подклассов требование, которое трассируется к методу, реализованному в классе, должно также трассироваться к методу в его подклассах, если этот метод переопределяется в подклассе. Это дополняет трассируемость требований, специфичных для самого подкласса [3].

15.5.3 Верификация

DO-332 добавляет или изменяет четыре мероприятия по верификации: (1) согласованность иерархии классов с требованиями, (2) локальная согласованность типов, (3) согласованность управления памятью с архитектурой и требованиями и (4) согласованность управления исключениями с архитектурой и требованиями.

DO-332 также добавляет мероприятие по тестированию в нормальном диапазоне, чтобы «убедиться, что конструкторы классов должным образом инициализируют состояние своих объектов и что начальное состояние согласуется с требованиями к классу» [3].

В DO-332 были добавлены две цели верификации, специфичные для объектно-ориентированной технологии и связанных с ней методов:

- *Цель ОО10 из таблицы ОО.А-7 документа DO-332: «Проверить локальную согласованность типов» [3]. Мероприятия для этой цели приведены в разделе ОО.6.7 документа DO-332.*
- *Цель ОО11 из таблицы ОО.А-7 документа DO-332: «Проверить робастность использования динамического управления памятью» [3]. Мероприятия для этой цели приведены в разделе ОО.6.8 документа DO-332.*

15.5.4 Уязвимости

Уникальной особенностью DO-332 является концепция уязвимостей. Приложение ОО.D документа DO-332 содержит перечень уязвимостей, которые могут присутствовать в системе, основанной на объектно-ориентированной технологии, или в системе, использующей связанные с ней методы. Уязвимости разделены на две категории: (1) ключевые особенности и (2) общие вопросы. Приложение ОО.D является важной частью DO-332 и содержит ценные указания по решению некоторых из наиболее сложных вопросов, связанных с объектно-ориентированной технологией и связанными с ней методами.

Уязвимости для *ключевых особенностей* определены в разделе ОО.D.1 документа DO-332 вместе с кратким изложением поддерживающих указаний и мероприятий. Рассматриваются следующие ключевые особенности: наследование, параметрический полиморфизм, перегрузка, преобразование типов, управление исключениями, динамическое управление памятью и виртуализация.

Уязвимости для *общих вопросов* объектно-ориентированной технологии и связанных с ней методов рассматриваются в разделе ОО.D.2 документа DO-332. Эти вопросы не ограничиваются объектно-ориентированной технологией, именно поэтому в названии дополнения присутствуют слова *related techniques* (связанные методы), и могут встречаться и в системе без объектно-ориентированной технологии. В разделе ОО.D.2 документа DO-332 описано, почему трассируемость, структурное покрытие, использование компонентов и анализ ресурсов могут быть более сложными для объектно-ориентированной технологии и связанных с ней методов, а также приведены дополнительные указания, которые следует учитывать при использовании этого подхода.

Уязвимости из приложения ОО.D документа DO-332 отсылают к указаниям из разделов ОО.4-ОО.12 дополнения, где определены цели и мероприятия, обеспечивающие учет и устранение этих уязвимостей.

15.5.5 Безопасность типов

Одной из уникальных особенностей DO-332 является выделение *безопасности типов* как средства смягчения ряда уязвимостей в системах, основанных на объектно-

ориентированной технологии и контроля уровня и глубины тестирования, необходимых для полной верификации системы, основанной на объектно-ориентированной технологии. Безопасность типов связана с поведением между классами и подклассами. В DO-332 сказано: «Чтобы тип был корректным подтипом некоторого другого типа, он должен демонстрировать то же поведение, что и каждый из его супертипов. Иными словами, любой подтип данного типа должен быть пригоден к использованию везде, где требуется данный тип» [3]. Для задания и обеспечения безопасности типов DO-332 опирается на принцип подстановки Лисков (Liskov Substitution Principle, LSP). Этот принцип определяет, что считается корректным подклассом, то есть подклассом, безопасным по типам, и тем самым ограничивает допустимое поведение подкласса. DO-332 поясняет:

Принцип формулируется так: «Пусть $q(x)$ – свойство, доказуемое для объектов x типа T . Тогда $q(y)$ должно быть истинно для объектов y типа S , где S является подтипом T ». ... Каждая подпрограмма, переопределенная в подклассе, должна удовлетворять следующим требованиям, предъявляемым к той же подпрограмме в любом из суперклассов этого подкласса:

- Предусловия не могут быть усилены, постусловия не могут быть ослаблены, а инварианты не могут быть ослаблены [3].

С точки зрения целей верификации на основе требований, определенных в DO-178C, соответствие этому принципу означает, что любой подтип, или подкласс, данного типа, или родительского класса, должен удовлетворять всем требованиям данного типа, или родительского класса. Для подтверждения соблюдения этого принципа можно использовать формальные методы или тестирование [3].

15.5.6 Связанные методы

Как отмечалось ранее, DO-332 содержит специальные указания по динамическому управлению памятью и виртуализации. Эти темы связаны с объектно-ориентированной технологией, но не являются специфичными только для нее. Указания по динамическому управлению памятью задают подход к спецификации, проектированию и оценке системы управления памятью для предсказуемого использования. Указания по виртуализации важны для интерпретаторов и технологий гипервизоров.

15.5.7 Часто задаваемые вопросы

DO-332 включает 39 часто задаваемых вопросов (Frequently asked questions, FAQ) по объектно-ориентированной технологии и связанным с ней методам, а также по указаниям, приведенным в дополнении. Эти часто задаваемые вопросы помогают уточнить замысел дополнения и некоторые из наиболее сложных технических тем, связанных с объектно-ориентированной технологией. Эти вопросы разделены на пять категорий: (1) общие вопросы, (2) вопросы, связанные с требованиями, (3) вопросы, связанные с проектирова-

нием, (4) вопросы, связанные с языками программирования, и (5) вопросы, связанные с верификацией.

15.6 Рекомендации по объектно-ориентированной технологии

При рассмотрении возможности использования объектно-ориентированной технологии в критичных по безопасности приложениях предлагаются следующие рекомендации:

Рекомендация 1: Рассмотрите DO-332, чтобы убедиться, что технические сложности, связанные с объектно-ориентированной технологией, понятны. Это желательно сделать до принятия окончательного решения об использовании объектно-ориентированной технологии.

Рекомендация 2: Если объектно-ориентированная технология для вас в новинку, начните с небольшого и менее критичного проекта (например, с ПО уровня C или D либо с инструмента) и затем переходите к более крупному, более сложному и/или более критичному ПО. Это дает возможность проработать как некоторые процессные вопросы, так и технические сложности.

Рекомендация 3: Следуйте указаниям DO-332. Сертифицирующие органы и авиационная отрасль более 10 лет исследовали объектно-ориентированную технологию, выявляли проблемы и вырабатывали обоснованные подходы к их устранению. DO-332 является итогом этих усилий.

Рекомендация 4: Подготовьте сопоставление с указаниями DO-332, чтобы убедиться, что указания и выявленные уязвимости полностью учтены. Как отмечалось ранее, уязвимости являются критически важной частью дополнения по объектно-ориентированной технологии и связанным с ней методам.

*Рекомендация 5: Ожидайте определенных трудностей. Внедрение новой технологии впервые может оказаться минным полем неожиданных проблем. Вместо *leading edge* (передовой рубеж) это иногда называют *bleeding edge* (слишком передовой, до крови). Можно надеяться, что усилия, вложенные сертифицирующими органами и отраслью за последние годы, позволили выявить основную массу проблем, но некоторые сюрпризы, специфичные для конкретного проекта, несомненно, все равно возникнут.*

Рекомендация 6: Тесно координируйтесь с сертифицирующими органами. Поскольку объектно-ориентированная технология в авиационной отрасли все еще остается сравнительно новой, сертифицирующие органы обычно внимательно за ней наблюдают. Можно надеяться, что по мере накопления положительного опыта необходимость в столь интенсивном надзоре уменьшится. До тех пор сертифицирующие органы будут хотеть знать детали проекта на основе объектно-ориентированной технологии.

Рекомендация 7: Оцените, применимы ли эти указания к каким-либо проектам, не использующим объектно-ориентированную технологию. Как отмечалось ранее, часть указаний DO-332 может быть применима и к проектам без объектно-ориентированной технологии. В частности, новые технологии, использующие динамическое управление памятью или виртуализацию, вероятно, потребуют применения указаний DO-332.

15.7 Заключение

С учетом технического прогресса последних лет, а также появления четких указаний по объектно-ориентированной технологии в документе DO-332, дверь для объектно-ориентированной технологии в авиации теперь открыта. Вероятно, она станет более обычным явлением в авиационном ПО.

Ссылки

1. Federal Aviation Administration, *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, Vol. 1 (October 2004).
2. ANSI/IEEE Standard, *Glossary of Software Engineering Terminology* (1983).
3. RTCA DO-332, *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
4. Aerospace Vehicle Systems Institute, *Guide to the Certification of Systems with Embedded Object-Oriented Software* (2001).
5. Federal Aviation Administration, *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, Vols. 1-4 (October 2004).
6. J. J. Chilenski, T. C. Timberlake, and J. M. Masalskis, *Issues Concerning the Structural Coverage of Object-Oriented Software*, DOT/FAA/AR-02/113 (Washington, DC: Office of Aviation Research, November 2002).
7. J. J. Chilenski and J. L. Kurtz, *Object-Oriented Technology Verification Phase 2 Report-Data Coupling and Control Coupling*, DOT/FAA/AR-07/52 (Washington, DC: Office of Aviation Research, August 2007).
8. J. J. Chilenski and J. L. Kurtz, *Object-Oriented Technology Verification Phase 2 Handbook-Data Coupling and Control Coupling*, DOT/FAA/AR-07/19 (Washington, DC: Office of Aviation Research, August 2007).
9. J. J. Chilenski and J. L. Kurtz, *Object-Oriented Technology Verification Phase 3 Report-Structural Coverage at the Source-Code and Object-Code Levels*, DOT/FAA/AR-07/20 (Washington, DC: Office of Aviation Research, August 2007).
10. J. J. Chilenski and J. L. Kurtz, *Object-Oriented Technology Verification Phase 3 Handbook-*

Рекомендуемое чтение[*]

1. M. Weisfeld, *The Object-Oriented Thought Process* (Indianapolis, IN: SAMS Publishing, 2000): Эта книга дает простое введение в основы объектно-ориентированной технологии. Это хороший ресурс для тех, кто переходит от функционального подхода к объектно-ориентированной технологии.
2. G. Booch, *Object-Oriented Analysis and Design*, 2nd edn. (Reading, MA: Addison-Wesley, 1994): Эта книга дает практическое введение в концепции, методы и применение объектно-ориентированной технологии.
3. B. Webster, *Pitfalls of Object-Oriented Development* (New York: M&T Books, 1995): Хотя книга уже несколько устарела, она дает добротный обзор потенциальных проблем разработки на основе объектно-ориентированной технологии.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1995): Шаблоны широко используются сообществом объектно-ориентированной технологии для решения задач анализа и проектирования. Эта книга дает руководство по эффективной разработке и использованию шаблонов.
5. B. Meyer, *Object-Oriented Software Construction*, 2nd edn. (Upper Saddle River, NJ: Prentice Hall, 1997): Эта объемная книга дает хорошую фундаментальную информацию для разработчиков, работающих с объектно-ориентированной технологией.
6. R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Reading, MA: Addison-Wesley, 2000): Эта книга посвящена тестированию объектно-ориентированной технологии, а это один из наиболее трудных ее аспектов.

*AVSI – это исследовательский консорциум в аэрокосмической отрасли, цели которого состоят в снижении затрат и сохранении безопасности сложных подсистем летательных аппаратов.

*Исследование возглавлял John Joseph Chilenski из Boeing Company. В число других исследователей от Boeing входили John L. Kurtz, Thomas C. Timberlake и John M. Masalskis. Отчеты и справочники доступны на сайте FAA (www.faa.gov).

*Эти источники также указаны в *OOTiA Handbook*, том 1. Хотя с момента публикации справочника прошло уже несколько лет, именно к этим источникам автор по-прежнему обращается, когда требуется разобраться в вопросе объектно-ориентированной технологии.

Глава 16. DO-333 и формальные методы

Сокращения

Сокращение	Расшифровка
CFD	Вычислительная гидродинамика
DARPA	Агентство перспективных исследовательских проектов Министерства обороны США
FAA	Федеральное управление гражданской авиации США
TBU	Требование высокого уровня к ПО
IEC	Международная электротехническая комиссия
THU	Требование низкого уровня к ПО
ISO	Международная организация по стандартизации
NASA	Национальное управление по аэронавтике и исследованию космического пространства
SCM	Управление конфигурацией ПО
SQA	Гарантия качества ПО

16.1 Введение в формальные методы

Инженерное дело в целом в значительной степени опирается на математические модели, позволяющие выносить обоснованные суждения о проектах. Однако разработка программного обеспечения традиционно была менее формализованной, а временами даже довольно бессистемной, с тестированием в конце проекта для подтверждения того, что все получилось хорошо. Цель формальных методов (formal methods) состоит в том, чтобы перенести ту же математическую основу, которая используется в других инженерных дисциплинах, в цифровой мир, то есть и в ПО, и в программируемую аппаратуру. Формальные методы применяют логику и дискретную математику для моделирования «поведения системы и формального подтверждения того, что проект и реализация системы удовлетворяют функциональным свойствам и свойствам безопасности» [1].

Хотя я не являюсь специалистом по формальным методам, мне доводилось сталкиваться с ними более 15 лет, и я глубоко уважаю как саму техническую дисциплину формальных методов, так и ее огромный потенциал по повышению качества и безопасности цифровых систем.

Формальные методы уже много лет продвигаются академическим сообществом. Некоторые из самых умных, образованных и увлеченных людей, которых я знаю, занимаются именно формальными методами. Они настойчиво вели свои исследования и последовательно от-

стаивали свои убеждения. Приятно видеть, что плоды их многолетней работы начинают соединяться с прикладной инженерией.

На сегодняшний день формальные методы получили большее применение в мире электронной аппаратуры, главным образом потому, что аппаратные инструменты более стандартизованы и стабильны, что позволяет встраивать формальные методы в повседневную работу инженеров. Программные инструменты и технологии, такие как методологии работы с требованиями и проектированием, языки и зависимости от целевого вычислителя, по-прежнему быстро меняются. Программная инженерия еще не достигла того уровня зрелости, который характерен для электронной аппаратуры и других инженерных дисциплин.

Дэниел Джексон и соавторы хорошо подытожили состояние формальных методов в программной инженерии. Они объясняют, что традиционная разработка программного обеспечения опирается на ручной анализ и тестирование для целей валидации и верификации. Формальные методы тоже используют тестирование, но, кроме того, применяют нотации и языки для строгого анализа. Формальные методы также используют инструменты для рассуждения о свойствах требований, проектов и кода. «Практики относились к формальным методам скептически с точки зрения их практической применимости. Однако все больше свидетельств того, что формальные методы могут позволять создавать системы с очень высокой надежностью при разумных затратах...» [2].

Некоторые стандарты, такие как ISO/IEC[*] 15408, *Common Criteria for Information Technology Security Evaluation*, и британский стандарт Defence Standard 00-55, *Requirements for Safety Related Software in Defence Systems*, требуют как минимум некоторого применения формальных методов для наиболее критичных уровней.[†] В DO-178B формальные методы были признаны приемлемым альтернативным средством. Однако в мире гражданской авиации формальные методы применялись крайне мало. Некоторые поставщики авионики и инструментов начинают использовать формальные методы, и ожидается, что их применение будет расти. Похоже, что практический опыт и поддерживающие инструменты для формальных методов наконец достигли уровня зрелости, достаточного для использования в авиационной отрасли.

Мое первое знакомство с формальными методами получилось довольно забавным. Меня попросили выступить с обзором DO-178B и политики и руководящих указаний Федерального управления гражданской авиации США (FAA), относящихся к программному обеспечению, на встрече Агентства перспективных исследовательских проектов Министерства обороны США (DARPA) в районе Вашингтона, округ Колумбия. Я уже много раз выступала с такими докладами, поэтому просто заменила титульный слайд, добавила немного фоновой информации в свою презентацию PowerPoint, доехала на метро до Кристал-Сити

и прибыла выступать. Мое выступление было первым в повестке дня, поэтому вскоре после прибытия меня представили, и я начала презентацию. Примерно через 5 минут из 60 я начала замечать странные выражения на лицах слушателей. В аудитории было около 50 человек, и раньше я никого из них не встречала. Некоторые выглядели озадаченными. Многие выглядели раздраженными или просто откровенно сердитыми. Некоторые выражения лиц вообще не поддаются описанию. Я понимала, что задела какую-то чувствительную тему, но не имела ни малейшего представления, в чем дело, поэтому просто продолжала говорить. Над моими шутками никто не смеялся, и я перестала пытаться добавлять юмор. И только когда настало время вопросов и ответов, я узнала «остальную часть истории». Я выступала перед группой специалистов по формальным методам, у которых не было опыта реального мира гражданской авиации. Я же, со своей стороны, сертифицировала несколько воздушных судов, но не имела опыта в формальных методах. Так два мира и столкнулись. Я отделалась относительно легко и была рада вернуться тем же днем в свой привычный офис. С тех пор я узнала о формальных методах намного больше и возлагаю большие надежды на их роль в будущем ПО, критичного по безопасности.

16.2 Что такое формальные методы?

В документе 2002 года, опубликованном Национальным управлением по авиации и исследованию космического пространства (NASA) под названием *NASA Langley's research and technology-transfer program in formal methods*, формальные методы описываются следующим образом:

Формальные методы означают использование методов логики и дискретной математики при спецификации, проектировании и построении компьютерных систем, как аппаратных, так и программных, и опираются на дисциплину, требующую явного перечисления всех предположений и всех шагов рассуждения. Каждый шаг рассуждения должен быть экземпляром относительно небольшого числа допустимых правил вывода. По сути, верификация системы сводится к вычислению, которое может быть проверено машиной. В принципе, эти методы способны давать безошибочный проект, однако для этого требуется полная верификация от требований до реализации, что на практике делается редко. Таким образом, формальные методы представляют собой прикладную математику инженерии компьютерных систем. Они играют в проектировании компьютеров роль, аналогичную той, какую вычислительная гидродинамика (CFD) играет в авиастроительном проектировании, предоставляя средство расчета и тем самым предсказания того, каким будет поведение цифровой системы до ее реализации.

Огромный потенциал формальных методов теории признают уже давно, но сами формальные техники, за немногими исключениями, оставались уделом сравнительно узкого

круга академических специалистов... Важно понимать, что формальные методы не являются подходом «все или ничего» [1].

Как пишет Джон Рашби: «Формальные методы позволяют предсказывать свойства компьютерной системы на основе математической модели системы при помощи процесса, аналогичного вычислению» [3].

Документ DO-333 под названием *Formal Methods Supplement to DO-178C (KT-178C) and DO-278A* был недавно опубликован RTCA. Глоссарий DO-333 определяет формальные методы так: «Описательная нотация и аналитические методы, используемые для составления, разработки и обоснования математических моделей поведения системы. Формальный метод представляет собой формальный анализ, выполняемый на формальной модели» [4]. Иными словами, формальный метод – это формальная модель плюс формальный анализ. *Формальная модель* – это «абстрактное представление заданного набора аспектов системы, используемое для анализа, имитации, генерации кода либо их сочетания. Формальная модель – это модель, определенная с помощью формальной нотации» [4].[*] *Формальная нотация* – это «нотация, имеющая точный, однозначный, математически определенный синтаксис и семантику» [4]. *Формальный анализ* – это «использование математического рассуждения для гарантии того, что свойства всегда выполняются для формальной модели» [4].

Формальная модель обычно используется на этапе разработки проекта для установления различных свойств. В разделе FM.1.6.1 документа DO-333 приводятся примеры формальных моделей, включая графические модели, текстовые модели и абстрактные модели. Такие модели используют математически определенные синтаксис и семантику [4].

В настоящее время формальные модели обычно используются лишь для моделирования части поведения программного обеспечения, поскольку входные данные в виде требований высокого уровня к ПО «могут включать свойства, которые невозможно верифицировать формальным методом. Кроме того, модели могут быть недостаточно детализированы для содержательного анализа одних свойств и при этом вполне достаточны для других» [4].

Формальные модели могут использоваться для описания определенных свойств с высокой степенью достоверности, тем самым поддерживая безопасность. Во многих случаях модели обеспечивают лишь некоторые свойства. Поэтому важно определить границы модели. Свойства, находящиеся вне этих границ, рассматриваются другими моделями либо традиционным подходом DO-178C [4].

Формальные модели могут приносить пользу процессу разработки программного обеспечения, однако наиболее ценным аспектом формальных методов является именно формальный анализ формальных моделей. Формальный анализ используется для доказательства или га-

рантирования того, что ПО соответствует требованиям. Чтобы доказать или гарантировать соответствие требованиям, определяется набор свойств ПО, который либо создается специально, либо уже встроен в инструмент формального анализа. Когда набор свойств ПО полностью определяет набор требований, формальный анализ может использоваться для доказательства того, что этот набор свойств действительно выполняется [4].

Метод анализа может считаться формальным только в том случае, если устанавливаемое им свойство определяется корректно (sound). DO-333 объясняет это следующим образом:

Корректный анализ означает, что метод никогда не утверждает, будто свойство истинно, когда это не так. Обратный случай, когда утверждается, что свойство ложно, хотя оно может быть истинным, то есть то, что обычно называют «ложными тревогами», является вопросом удобства применения, а не вопросом корректности. Кроме того, допустимо, чтобы метод возвращал «неизвестно» или вообще не давал ответа при попытке установить, выполняется ли свойство; в таком случае требуется дополнительная верификация [4].

В DO-333 выделяются три типичные категории формального анализа [4]:

1. *Дедуктивные методы*, использующие математические аргументы для установления каждого свойства формальной модели. Доказательства обычно строятся с использованием инструмента доказательства теорем, чтобы показать корректность свойств программного обеспечения.
2. *Проверка модели (model checking)*, которая «исследует все возможные варианты поведения формальной модели, чтобы определить, выполняется ли заданное свойство».
3. *Абстрактная интерпретация*, которая строит консервативные представления семантики языка программирования для «корректного определения динамических свойств программ с бесконечным числом состояний... Ее можно рассматривать как частичное исполнение компьютерной программы, которое определяет определенные эффекты программы, например структуру управления, потоки информации, размер стека или число тактов, не выполняя при этом всех вычислений».

Эти три категории формального анализа имеют пару общих признаков. Во-первых, они опираются на формальные модели. «Соответствие между артефактами никогда не может быть продемонстрировано между неформальной моделью и формальной моделью с помощью формального анализа. Например, чтобы с помощью формальных методов продемонстрировать соответствие между спецификацией и кодом, обе стороны должны быть формальными моделями. Во-вторых, все виды формального анализа обычно реализуются с использованием инструмента» [4]. Следовательно, инструменты, применяемые для формального анализа, могут требовать квалификации. Сведения о квалификации инструментов приведены в главе

16.3 Потенциальные преимущества формальных методов

Формальные методы дают ряд потенциальных преимуществ, рассмотренных в этом разделе. По мере накопления опыта и развития технологий эти преимущества, вероятно, будут возрастать.

Потенциальное преимущество 1: Улучшение определения требований. Благодаря формальному характеру формальных моделей они могут обеспечивать полноту требований и более тщательную их проработку. Традиционные требования часто бывают неполными, неточными, неоднозначными и противоречивыми. Рашби пишет: «Многие проблемы при проектировании программного и аппаратного обеспечения вызваны неточностью, двусмысленностью, неполнотой, недопониманием и просто ошибками в формулировке требований высокого уровня, в описании промежуточных проектов или в спецификациях компонентов и интерфейсов. Некоторые из этих проблем можно объяснить трудностью описания крупных и сложных артефактов естественным языком» [3]. Использование формальной модели помогает выявлять такие слабые места раньше.

Потенциальное преимущество 2: Снижение числа ошибок. Формальные методы помогают меньше полагаться на человеческую интуицию и суждение, используя более строгий и математически обоснованный подход. За счет строгости, необходимой для определения формальной модели, можно уменьшить количество ошибок в требованиях и/или проекте.

Потенциальное преимущество 3: Выявление большего числа ошибок. Формальный анализ способен находить ошибки, которые при традиционных средствах верификации могли бы вообще остаться незамеченными. Он может вскрывать ошибки, недопонимание и тонкие неожиданные свойства, которые могли бы ускользнуть при традиционном подходе, основанном на рассмотрениях и тестировании.

Потенциальное преимущество 4: Повышение уверенности в свойствах безопасности. Использование формальных методов для моделирования и верификации характеристик безопасности системы может повысить уверенность в ее безопасности и качестве.

Потенциальное преимущество 5: Повышение уверенности для высокосложных систем. Формальные методы позволяют намного полнее анализировать пространство входов и выходов. Для очень сложных функций формальные методы могут проверять функциональность более полно, чем традиционное тестирование, которое охватывает лишь часть пространства входов и выходов.

Потенциальное преимущество 6: Повышение качества в областях, подверженных ошибкам. Хотя применение формальных методов ко всему проекту ПО может быть непрактич-

ным, их можно эффективно использовать в тех областях, где ошибки возникают чаще всего. Как правило, это и есть самые сложные области.

Потенциальное преимущество 7: Более эффективная реализация инструментов. Коммерческие инструменты, например автоматические генераторы кода, разрабатываются для использования несколькими производителями авионики. Если при создании таких инструментов применяются формальные методы, это может повысить робастность инструмента и снизить объем дополнительной верификации, требуемой от его пользователя. Например, автоматический генератор кода, разработанный с применением формальных методов, может уменьшить либо вообще устранить необходимость в анализе структурного покрытия для выходных результатов инструмента, поскольку формальные методы обеспечивают, что инструмент не будет порождать неточный, нетрассируемый, неполный или посторонний код.

Потенциальное преимущество 8: Снижение стоимости. Хотя применение формальных методов может потребовать большей строгости и больше ресурсов на начальном этапе, оно дает возможность находить ошибки раньше и с большей определенностью. Это может снизить общую стоимость разработки программного обеспечения, поскольку позднее выявление ошибок обходится весьма дорого.

Потенциальное преимущество 9: Сопровождаемость программного обеспечения. Поскольку формальные методы способны приводить к более чистым требованиям и архитектуре, они могут облегчить сопровождение системы. Стоит отметить, что даже при использовании формальных методов при повторном использовании или модификации существующего ПО все равно требуется осторожность.

16.4 Трудности формальных методов

При внедрении формальных методов проекты сталкиваются с рядом трудностей.

Трудность 1: недостаток подготовки порождает высокий уровень страха. Большинство инженеров-программистов и их руководителей все еще недостаточно знакомы с формальными методами и даже побаиваются их.

Обозначения выглядят непривычно, и у многих инженеров может не быть достаточной математической подготовки для применения таких инструментов. Эту трудность можно преодолеть с помощью обучения и удобных для пользователя инструментов.

Трудность 2: формальные методы применимы не ко всем задачам. У формальных методов есть свои пределы. Важно использовать их там, где они наиболее эффективны и где у них наибольший потенциал, например для сложных или проблемных функций.

Трудность 3: бывает сложно сочетать формальные и не столь формальные методы. По-

скольку большинство проектов применяет формальные методы только к части общих мероприятий по определению требований и верификации, их необходимо интегрировать с более традиционным процессом. Возможны ситуации, когда формальные методы используются совместно с другими не вполне традиционными подходами, такими как объектно-ориентированная технология или разработка на основе моделей. Чтобы справиться с этой трудностью, важно видеть общую картину и помнить о целях, которые определяют соответствующие действия.

Трудность 4: ограниченное число доступных специалистов по формальным методам. Практиков формальных методов не так уж много, и это может стать препятствием для успешного применения этих техник. Как отмечалось в трудности 1, обучение и удобные инструменты тоже помогают смягчить эту проблему.

Трудность 5: формальные методы могут требовать значительных ресурсов. Формальные методы требуют специальных навыков и инструментов. Кроме того, их применение само по себе может быть весьма непростым. По этой причине лучше использовать их в наиболее критичных, сложных и подверженных ошибкам областях, где отдача от вложений будет максимальной.

Трудность 6: чрезмерная уверенность в верификационных возможностях формальных методов. Иногда ошибочно считают, что формальные методы могут гарантировать корректность. Они действительно дают более высокую уверенность в соответствии программного обеспечения требованиям, однако, как объясняют Боуэн и Хинчи: «Если организация создала не ту систему, которая нужна (валидация), никакая безупречность при ее правильной реализации (верификация) не сможет исправить эту ошибку» [5].

Трудность 7: существуют ложные представления о тестировании. Некоторые утверждают, что формальные методы означают отсутствие необходимости в тестировании. Формальные методы могут помочь уменьшить вероятность некоторых ошибок или выявить их, но они должны сопровождаться надлежащим тестированием [5]. В DO-333 подчеркивается необходимость тестировать программное обеспечение даже при применении формальных методов. В частности, требуется тестирование на целевом вычислителе для проверки интеграции аппаратуры и ПО.

Трудность 8: руководство отрасли и сертификационные указания пока еще не стандартизованы. Если попросить трех человек объяснить, что такое формальные методы, можно получить как минимум четыре ответа. Завершение DO-333 стало большим шагом к стандартизации, однако множество вопросов по-прежнему приходится решать отдельно в каждом проекте.

Трудность 9: поддержка инструментами все еще неполная. Инструменты формальных ме-

тодов за последние несколько лет заметно улучшились, но для того, чтобы они стали практически пригодными для авиационного сообщества, еще предстоит большая работа. Инструменты должны естественно встраиваться в ЖЦ ПО и быть удобными для предметных специалистов. Если инструменты непрактичны для тех, кто обладает доменной экспертизой, например для инженера по системам управления полетом, они не будут использоваться эффективно, а перечисленные выше потенциальные преимущества не реализуются.

16.5 Обзор DO-333

16.5.1 Назначение DO-333

DO-333 дополняет DO-178C путем изменения, удаления и добавления целей, относящихся именно к формальным методам. Дополнение DO-333 основано на следующих ключевых принципах [6]:

- Формальный метод представляет собой применение формального анализа к формальной модели.
- Формальная модель должна быть записана в нотации с математически определенными синтаксисом и семантикой.
- Формальные методы могут применяться на различных этапах верификации в ЖЦ ПО, для всего этапа или его части и для всей разрабатываемой системы или ее части.
- Формальный метод никогда не должен выдавать результат, который может оказаться неверным, то есть формальный анализ должен быть корректным.
- Результаты формального анализа исходного кода можно распространить на соответствующий объектный код, если достаточно подробно понимать процессы компиляции, компоновки и загрузки.
- Тестирование всегда необходимо для обеспечения совместимости программного обеспечения с целевой аппаратурой и для полной проверки понимания связи между исходным и объектным кодом.

16.5.2 Сравнение DO-333 и DO-178C

16.5.2.1 Планирование и разработка

DO-333 уточняет процессы планирования и разработки из DO-178C для случаев использования формальных методов. В DO-333 поясняется, что необходимо разработать планы и стандарты, описывающие и учитывающие подход к формальным методам. Если формальная модель используется при разработке без формального анализа, применять DO-333 не требуется; в этом сценарии достаточно самого DO-178C. Иными словами, если на этапах требований, проектирования и/или кодирования используется формальная модель или несколько слоев моделей, применяются цели DO-178C. Дополнение вносит некоторые уточнения для

разработки, специфичные для моделей, но в целом это тот же процесс, что определен в DO-178C.

16.5.2.2 Управление конфигурацией, гарантия качества ПО и взаимодействие с сертифицирующим органом

Процессы управления конфигурацией ПО, гарантии качества ПО и взаимодействия с сертифицирующим органом из DO-178C в DO-333 не изменяются.

16.5.2.3 Верификация

Основные различия между DO-178C и DO-333 сосредоточены в области верификации. Поскольку формальный анализ способен доказывать или опровергать корректность формальной модели, некоторые традиционные цели DO-178C по рассмотрению, анализу и тестированию могут быть заменены формальным анализом. DO-333 модифицирует цели и мероприятия по рассмотрению и анализу требований высокого уровня к ПО, требований низкого уровня к ПО, архитектуры ПО и исходного кода так, чтобы они были специфичны для формальных методов. Формальный анализ может использоваться для выполнения целей, связанных с соответствием входов и выходов, точностью, согласованностью, совместимостью с целевым вычислителем, верифицируемостью, соответствием стандартам, трассируемостью, точностью алгоритмов и корректностью формализации требований [4].

Для верификации с использованием формальных методов существуют две основные трудности: 1) верификация исполняемого объектного кода (таблица A-6 DO-178C и таблица FM.A-6 DO-333) и 2) верификация верификации (таблица A-7 DO-178C и таблица FM.A-7 DO-333).

Сначала рассмотрим *верификацию исполняемого объектного кода*. На текущем этапе заменить тестирование исполняемого объектного кода формальным анализом невозможно. Формальный анализ может использоваться как дополнение к тестированию и для проверки соответствия требованиям, однако из-за зависимостей от целевого вычислителя модели пока еще недостаточны, чтобы заменить тестирование. Поэтому цели верификации исполняемого объектного кода одинаковы и при использовании формальных методов, и при использовании традиционных методов разработки программного обеспечения. То есть таблица FM.A-6 DO-333 содержит те же цели, что и таблица A-6 DO-178C. Однако в DO-333 была добавлена дополнительная цель для случая, когда формальный анализ используется для проверки свойств исполняемого объектного кода. В таблице FM.A-7 DO-333 цель FM9 сформулирована так: «Верификация сохранения свойств между исходным и объектным кодом» [4].

Проверяя корректность преобразования исходного кода в объектный, можно использовать формальный анализ, выполненный на уровне исходного кода по отношению к требованиям

высокого или низкого уровня к ПО, чтобы сделать вывод о корректности объектного кода по отношению к этим же требованиям. Это похоже на то, как метрики покрытия, полученные на исходном коде, могут использоваться для подтверждения достаточности тестов при верификации целевой системы [7].

Провести такой анализ, вероятно, будет непросто. Однако по мере развития технологий эмуляции и переносимости это может становиться все более осуществимым.

Вторая область трудностей – это *верификация верификации*. В таблице A-7 DO-178C содержится девять целей. В таблице 16.1 показано соответствие между целями таблицы A-7 DO-178C и целями таблицы FM.A-7 DO-333, которые заменяют цели DO-178C. Из таблицы видно, что цели 1-4 и 9 из DO-178C имеют эквиваленты в DO-333. Однако четыре цели DO-178C, связанные со структурным покрытием, а именно цели 5-8, заменены в DO-333 одной целью. Поскольку структурное покрытие связано с выполнением тестов и измерением покрытия кода, при использовании формальных методов предлагается альтернатива. Эта альтернатива по-прежнему должна удовлетворять замыслу структурного покрытия, то есть выявлять недостатки тестов, основанных на требованиях, обнаруживать неполноту требований и находить посторонний код, включая мертвый код, а также отключенный код. Для выполнения целей по структурному покрытию дополнение предлагает следующие четыре мероприятия [4,7]:

Таблица 16.1 Сравнение целей DO-178C и DO-333 для верификации верификации

Цель таблицы A-7 DO-178C	Цель таблицы FM.A-7 DO-333
1. Тестовые процедуры корректны.	FM1. Примеры и процедуры формального анализа корректны.
2. Результаты тестирования корректны, а расхождения объяснены.	FM2. Результаты формального анализа корректны, а расхождения объяснены.
3. Достигнуто покрытие требований высокого уровня к ПО тестами.	FM3. Достигнуто покрытие требований высокого уровня к ПО.
4. Достигнуто покрытие требований низкого уровня к ПО тестами.	FM4. Достигнуто покрытие требований низкого уровня к ПО.
5-8. Достигнуто покрытие структуры ПО тестами на уровне модифицированного покрытия условий/решений, покрытия решений, покрытия операторов и межкомпонентной связности по данным и по управлению.	FM5-FM8. Достигнуто покрытие верификацией структуры ПО. Это одна цель, заменяющая четыре цели по структурному покрытию из DO-178C.

Цель таблицы A-7 DO-178C	Цель таблицы FM.A-7 DO-333
9. Выполнена верификация дополнительного кода, который невозможно трассировать до исходного кода.	FM9. Выполнена верификация сохранения свойств между исходным и объектным кодом.
N/A	FM10. Формальный метод корректно определен, обоснован и адекватен поставленной задаче.

Источники: RTCA DO-333, Formal Methods Supplement to DO-178C and DO-278A, RTCA, Inc., Washington, DC, December 2011; RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification, RTCA, Inc., Washington, DC, December 2011.

- *Полное покрытие каждого требования.* Когда для формального анализа вводятся предположения, все эти предположения должны быть проверены, чтобы обеспечить полное покрытие каждого требования.
- *Полнота набора требований.* Для формально моделируемых требований необходимо показать, что набор требований полон относительно предполагаемых функций. Следует подтвердить, что для всех входных условий задан требуемый выход, а для всех выходов заданы требуемые входные условия. Если требования неполны, их необходимо актуализировать. Если полноту требований показать невозможно, требуется структурное покрытие.
- *Выявление непреднамеренных связей потоков данных.* Замысел здесь состоит в том, чтобы подтвердить, что поток данных в исходном коде соответствует требованиям и что между входами и выходами кода нет непреднамеренных зависимостей. Такие непреднамеренные зависимости необходимо устранять, например путем добавления требований или удаления ошибочного кода.
- *Выявление постороннего и отключенного кода.* Цель здесь та же, что и в DO-178C, в отношении постороннего кода, включая мертвый код, и отключенного кода. Глава 17 поясняет, что такое посторонний, мертвый и отключенный код.

Применение таблицы FM.A-7 из DO-333 может оказаться непростой задачей. Выбранный подход должен гарантировать, что весь непокрытый код будет выявлен. Кроме того, похоже, что руководство DO-333 не затрагивает анализ потока управления. Тем, кто внедряет формальные методы, потребуется выявлять непреднамеренный поток управления так же, как и непреднамеренный поток данных. При использовании формальных методов важно тесно координировать с сертифицирующими органами выбранный подход к выполнению

целей таблицы FM.A-7 DO-333.

Поскольку формальные методы обычно применяются только к части программного обеспечения, а не ко всему объему, выбранный подход должен четко указывать, где используется DO-333, а где применяется DO-178C и/или другое дополнение.

16.6 Другие ресурсы

Эта глава дала лишь введение в формальные методы. По формальным методам существует множество книг, отчетов и статей. Есть люди, которые посвятили этой теме многие годы своей жизни. Они весьма продуктивно создавали отчеты и другие материалы, чтобы просвещать коллег. Если вы решите использовать формальные методы в проекте, тему придется изучить гораздо глубже и, вероятно, привлечь специалистов, которые помогут вашей команде и обучат ее. В разделе «Ссылки» перечислены некоторые ресурсы, которые я считаю наиболее полезными. Каждая из этих ссылок, в свою очередь, ведет к другим полезным материалам.

Ссылки

1. R. W. Butler, V. A. Carreno, B. L. Di Vito, K. J. Hayhurst, C. M. Holloway, J. M. Maddalon, P. S. Miner, C. Munoz, A. Geser, and H. Gottliebsen, NASA Langley's research and technology-transfer program in formal methods (Hampton, VA: NASA Langley Research Center, May 2002).
2. D. Jackson, M. Thomas, and L. I. Millett, *Software for Dependable Systems: Sufficient Evidence?* (Washington, DC: Committee on Certifiably Dependable Software Systems, National Research Council, 2007).
3. J. Rushby, Formal methods and the certification of critical systems, Technical Report CSL-93-7 (Menlo Park, CA: SRI International, December 1993).
4. RTCA DO-333, *Formal Methods Supplement to DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
5. J. P. Bowen and M. G. Hinchey, Ten commandments of formal methods... Ten years later, *IEEE Computer* January, 40-48, 2006.
6. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
7. D. Brown, H. Delseny, K. Hayhurst, and V. Wiels, Guidance for using formal methods in a certification context, ERTS Toulouse Conference (Toulouse, France, May 2010).
8. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).

*ISO/IEC означает International Standards Organization / International Electrical Technical Commission.

†Хотя стандарт Defence Standard 00-56 (*Safety Management Requirements for Defence Systems*), пришедший на смену Defence Standard 00-55, этого не требует. Однако Defence Standard 00-56 признает формальное доказательство и формальный анализ приемлемыми средствами представления свидетельств безопасности.

*DO-333 по существу определяет формальные методы так: *формальные методы = формальная модель + формальный анализ*. Обычно формальные методы так не описывают; такой подход был использован в DO-333, чтобы объяснить формальные методы через процессы жизненного цикла из DO-178C.

Часть V Специальные темы

В части III был дан обзор DO-178C (КТ-178С) и рассмотрены процессы, необходимые для соответствия ему, включая процессы планирования, разработки, верификации, управления конфигурацией, гарантии качества ПО и взаимодействия с сертифицирующим органом. В части IV были представлены руководство по квалификации инструментов (DO-330), а также технологические дополнения к DO-178C по разработке на основе моделей (DO-331), объектно-ориентированной технологии (DO-332) и формальным методам (DO-333). В части V рассматриваются технологии и специальные темы, актуальные для многих работ по разработке программного обеспечения, критичного по безопасности. Рассматриваются следующие темы:

- непокрытый код (мертвый, посторонний и отключенный код) (глава 17)
- загружаемое в полевых условиях программное обеспечение (глава 18)
- модифицируемое пользователем ПО (глава 19)
- операционные системы реального времени (глава 20)
- обособление ПО (глава 21)
- данные конфигурации (глава 22)
- аэронавигационные данные (глава 23)
- повторное использование программного обеспечения, включая ранее разработанное ПО и готовое коммерческое ПО (commercial off-the-shelf, COTS) (глава 24)
- обратная разработка (глава 25)
- аутсорсинг и офшорная разработка (глава 26)

Есть еще несколько тем, которые мне хотелось осветить в этой части, включая интегрированную модульную авионику, электронную бортовую аппаратуру, безопасность и электронные полетные планшеты. Однако из-за ограничений по времени и объему они не рассматриваются. Некоторые из этих тем, впрочем, планируется осветить в новом издании *Digital Avionics Handbook* издательства CRC Press.

Глава 17. Непокрытый код (мертвый, посторонний и отключенный код)

Сокращения

Сокращение	Расшифровка
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)

Сокращение	Расшифровка
SAS	Итоговый отчёт разработки ПО (Software Accomplishment Summary)
SDP	План разработки ПО (Software Development Plan)
SVP	План верификации ПО (Software Verification Plan)

17.1 Введение

Когда выполняется анализ структурного покрытия, чтобы убедиться, что тесты, основанные на требованиях, полностью охватывают структуру кода, как обсуждалось в главе 9, может оказаться, что некоторый код не *покрыт*, то есть не выполняется тестами. Если отсутствие покрытия вызвано отсутствующими требованиями или тестами, требования и тесты при необходимости обновляются и выполняются повторно. Может существовать код, который нельзя выполнить в тестовой среде, но который все же соответствует требованиям, например код, работающий во время инициализации, или защитный код. Такой код необходимо верифицировать альтернативным способом, например анализом или инспекцией кода. Однако даже после этого может остаться некоторый непокрытый код, классифицируемый как посторонний, мертвый или отключенный. В этой главе обсуждаются эти классы кода и даются рекомендации по обращению с ними.

17.2 Посторонний и мертвый код

Термин «посторонний код» (*extraneous code*) был добавлен при переходе от DO-178B к DO-178C. Вокруг того, как назвать этот класс кода, который 1) не выполняется во время тестирования, основанного на требованиях, и 2) не трассируется к функциональным требованиям, велось немало споров. Рассматривались разные альтернативные термины, например нетрассируемый код (*untraceable code*), недостижимый код (*unreached code*) и непокрытый код (*noncovered code*). В итоге в этом соревновании победил термин *extraneous code*.

DO-178C определяет посторонний код так: «Код (или данные), которые не трассируемы на требования к системе или к ПО. Примером постороннего кода является унаследованный код, который ошибочно был сохранен, хотя требования к нему и его тестовые примеры были удалены. Другим примером постороннего кода является мертвый код» [1].

Из определения постороннего кода следует, что мертвый код (*dead code*) является подмножеством постороннего кода. DO-178C поясняет, что мертвый код – это «исполняемый объектный код или данные, которые вследствие ошибки разработки ПО не могут быть выполнены (код) или использованы (данные) при любой нормальной конфигурации среды целевого вычислителя. Мертвый код не трассируется на системные требования и на требования к ПО...» [1]. Ниже приведен простой пример мертвого кода. Как видно, строка `calculate`

airspeed недостижима из-за того, как написан код.

```
if (UP and OVER) and not OUT
    calculate wheelspeed
    if OUT
        calculate airspeed
    end if
end if
```

DO-178C требует удалять посторонний код, включая мертвый код. Помимо самого удаления, выполняется анализ, позволяющий определить 1) последствия удаления кода и 2) необходимость повторной верификации до одобрения кода в составе бортовой системы [1]. Для многих проектов это руководство оказалось крайне неприятной пилюлей. Однако следует отметить, что DO-178C также приводит ряд распространенных исключений из подхода *вырывать с корнем*, включая следующее [1]:

- *встроенные идентификаторы* в коде не считаются посторонним или мертвым кодом. Примеры встроенных идентификаторов включают контрольные суммы и идентификационные обозначения.
- *Структуры безопасного программирования*, используемые для улучшения робастности, тоже не считаются посторонним или мертвым кодом. Однако такие практики защитного программирования должны быть четко определены в стандартах кодирования. Кроме того, необходимо показать, что защитная конструкция поддерживает реализацию требований и проекта. Например, часто требуемая практика защитного кодирования состоит в наличии ветви *else* для каждого оператора *if*.
- *Отключенный код* не является ни посторонним, ни мертвым кодом. Отключенный код будет рассмотрен далее в этой главе.
- *исходный или объектный код, отсутствующий в исполняемом объектном коде*, например из-за настроек компилятора или компоновщика, не считается посторонним или мертвым кодом. Однако требуется анализ, показывающий, что такой код действительно отсутствует в исполняемом объектном коде. Кроме того, должны существовать процедуры, гарантирующие, что такой код случайно не попадет в исполняемый объектный код в будущем, например процедуры сборки, явно задающие соответствующие настройки и объясняющие причину их существования.

Когда анализ структурного покрытия впервые выявляет непокрытый код, полученные данные следует изучить и проанализировать, чтобы определить, является ли это проблемой требований, проблемой тестового примера или проблемой кода. Не надо автоматически

предполагать, что непокрытый код является посторонним или мертвым. По моему опыту, большая часть первоначально обнаруженного непокрытого кода связана с отсутствующими требованиями, отсутствующими тестовыми примерами или ошибками логики. Когда установлено, что непокрытый код действительно является посторонним или мертвым, приходится тратить время на анализ влияния этого кода и определение наилучшего решения. Некоторый код может выглядеть *мертвым*, но стоит его удалить, и начинается хаос. В одной компании мне рассказывали, что часть своего кода они называли *кодом-зомби*, потому что он казался мертвым, но время от времени оживал, что для систем, критичных по безопасности, совершенно неприемлемо.

17.2.1 Как избежать позднего обнаружения постороннего и мертвого кода

Многие разработчики усвоили на горьком опыте, что быстрое удаление мертвого или постороннего кода на поздних стадиях жизненного цикла проекта может быть проблематичным. К удалению мертвого или постороннего кода после проведения формального тестирования программного обеспечения, то есть тестирования, дающего сертификационный зачет, следует относиться с предельной осторожностью. Если код ошибочно признан посторонним или мертвым и удален, может оказаться трудно определить последствия без существенной повторной верификации, возможно даже без повторного выполнения всех тестов. Поэтому важно прилагать все усилия, чтобы находить проблемы кодирования как можно раньше. Ниже приведены некоторые рекомендации, позволяющие избежать позднего обнаружения мертвого или постороннего кода:

- внедрять качественные рассмотрения кода, специально ориентированные на соответствие кода требованиям и его трассируемость к ним.
- поддерживать данные трассировки в актуальном состоянии. Каждый раз, когда меняются требования или код, данные трассировки должны обновляться.
- выполнять тесты и анализировать структурное покрытие как можно раньше. Как отмечалось в главе 9, выполнение тестов до рассмотрения тестовых примеров и процедур является типичной практикой. По мере изменения требований или кода тесты следует повторно запускать, а покрытие – анализировать заново. Эти ранние прогоны тестов и анализ покрытия помогают выявлять проблемы до финального прогона, который будет использоваться для сертификационного зачета, то есть до формального прогона, или прогона с зачетом (*for-score*).
- закладывать время на устранение проблем, выявленных во время тестирования. Как отмечалось в главе 9, многие проекты используют планирование, основанное на предположении об успехе, и исходят из того, что все тесты пройдут без проблем, но это подготавливает проект к провалу или как минимум к существенным задержкам.

- определить процессы и процедуры для работы с непокрытым кодом и включить примеры.

17.2.2 Оценка постороннего или мертвого кода

Даже после внедрения этих рекомендаций в программе все равно может поздно обнаружиться некоторый посторонний код, включая мертвый код. Если это происходит, рисунок 17.1 показывает шаги, которые обычно выполняются. Ниже описан каждый блок этой блок-схемы:

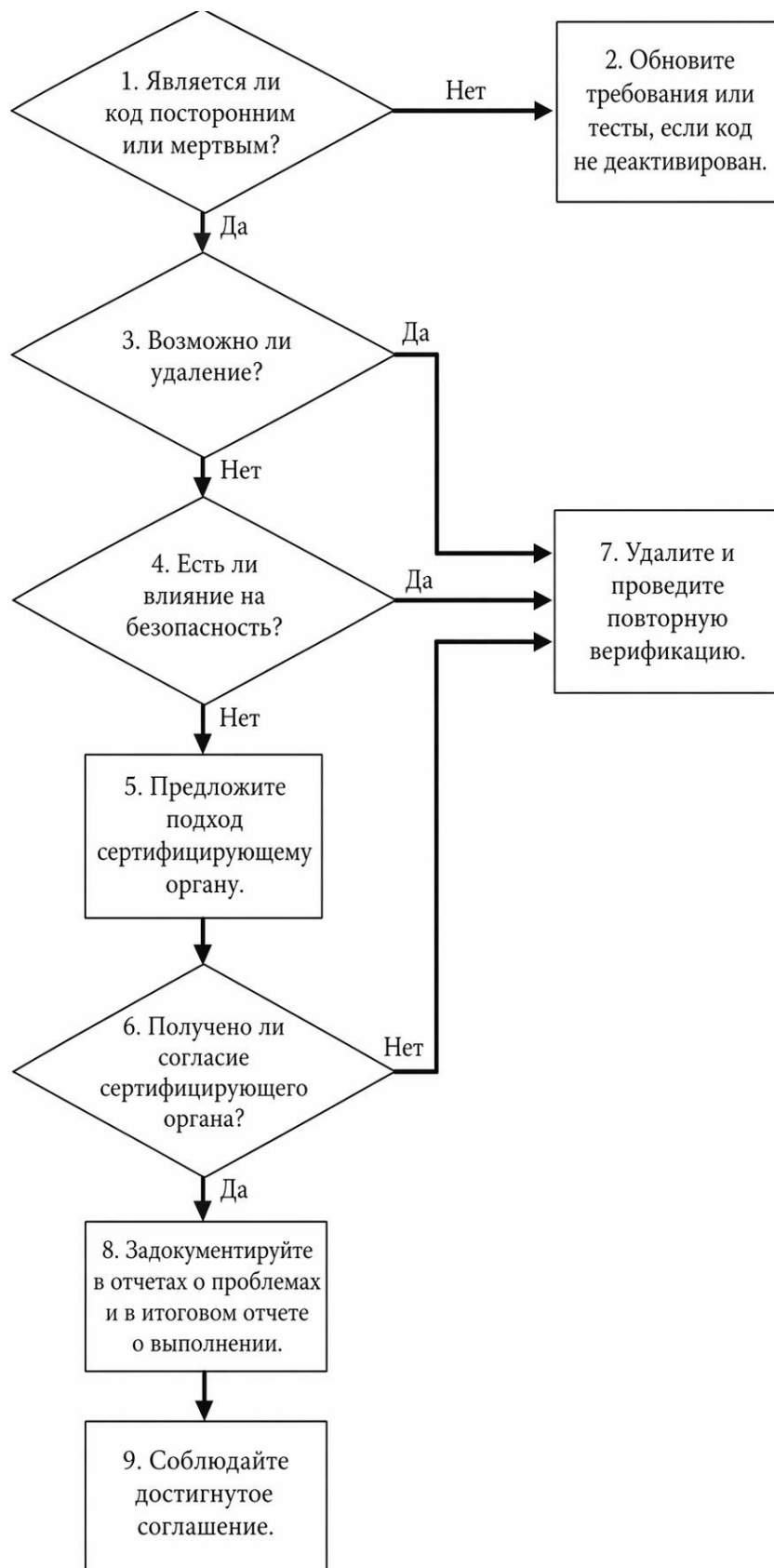


Рисунок 17.1 Шаги по оценке постороннего или мертвого кода.

1. Тщательно проанализировать непокрытый код, чтобы подтвердить, что он действительно является посторонним или мертвым.
2. Если непокрытый код не является посторонним, мертвым или отключенным, внести соответствующие изменения в требования или тесты. Если он является отключенным,

необходимо убедиться, что для него соблюдается руководство по отключенному коду, обсуждаемое в разделе 17.3.

3. Если непокрытый код является посторонним или мертвым, проанализировать возможность его удаления с учетом архитектуры. Одни фрагменты кода удалить легко, другие весьма сложно. При определении реализуемости может потребоваться учитывать и график работ. Посторонний или мертвый код, найденный поздно в программе, может иметь ограничения как по графику, так и технические ограничения.
4. Оценить последствия активации постороннего или мертвого кода, если его не удалять. Нужно рассмотреть, что произойдет, если такой код будет непреднамеренно активирован, то есть оценить последствия для безопасности и функциональности.
5. Если анализ с высокой степенью уверенности показывает, что непреднамеренная активация постороннего или мертвого кода безвредна с точки зрения безопасности, например в случае неиспользуемых переменных, может оказаться возможным получить специальное согласование от сертифицирующего органа. Такое согласование обычно требует удаления кода при первой постсертификационной модификации. Этот подход субъективен и должен быть тщательно проанализирован, обоснован и как можно раньше представлен сертифицирующему органу. На детали предлагаемой стратегии будут влиять критичность программного обеспечения и характер постороннего или мертвого кода.
6. Как можно раньше запросить согласование сертифицирующего органа по предлагаемой стратегии.
7. Удалить посторонний или мертвый код и провести повторную верификацию, включая повторный запуск соответствующих тестов, если выполняется хотя бы одно из следующих условий: 1) удаление реализуемо с технической точки зрения и по графику, 2) последствия активации не являются безвредными, то есть безопасность может пострадать, или 3) сертифицирующий орган не согласен одобрить программное обеспечение при сохранении постороннего или мертвого кода.
8. Если сертифицирующий орган согласен допустить наличие постороннего или мертвого кода при первоначальном одобрении, нужно задокументировать условия этого согласования и указать конкретные сообщения о проблемах, относящиеся к постороннему или мертвому коду, в итоговом отчёте разработки ПО (Software Accomplishment Summary, SAS).
9. Если согласование достигнуто, необходимо убедиться, что существуют шаги для его выполнения, то есть что график удаления действительно реализуется. Команда, кото-

рая будет модифицировать программное обеспечение, обычно не та же самая команда, которая его разрабатывала, должна быть уведомлена об этом согласовании при начале обновления ПО, чтобы обеспечить отслеживание изменений, их реализацию и верификацию.

Следует отметить, что обсуждение и оценка постороннего или мертвого кода обычно происходят на уровне исходного кода, поскольку большинство проектов измеряют структурное покрытие по исходному коду. Однако некоторые компании выполняют анализ структурного покрытия по объектному или машинному коду. В этом случае те же самые вопросы, связанные с посторонним или мертвым кодом, все равно должны быть решены, просто сам код выглядит иначе.

Как отмечалось в главе 9, для ПО уровня А существует дополнительная цель при выполнении анализа структурного покрытия по исходному коду. Анализируется трассируемость от исходного кода к объектному коду, чтобы убедиться, что компилятор не сгенерировал посторонний или мертвый код. Для ПО уровня А любой посторонний или мертвый код, сгенерированный компилятором, должен быть проанализирован и обработан так, как описано выше.

17.3 Отключенный код

Отключенный код часто используется для создания гибкого, но при этом полностью соответствующего требованиям программного обеспечения. Отключенный код связан с посторонним или мертвым кодом, потому что он может проявляться как непокрытый код при анализе структурного покрытия. Однако, в отличие от постороннего или мертвого кода, отключенный код планируется и реализуется намеренно.

Глоссарий DO-178C определяет отключенный код (*deactivated code*) следующим образом:

Исполняемый объектный код (или данные), трассируемый на требования и, в соответствии с проектом, либо (а) не предназначенный для исполнения (код) или использования (данные), например часть ранее разработанного компонента ПО, такая как неиспользуемый унаследованный код, неиспользуемые функции библиотеки или код, предназначенный для последующего наращивания, либо (б) исполняемый (код) или используемые (данные) только в конкретных конфигурациях среды целевого вычислителя, например код, который включается выбором контакта в аппаратуре или программируемой опцией ПО. Приведенные ниже примеры часто ошибочно классифицируются как отключенный код, хотя их следует рассматривать как необходимые для реализации проекта или требований: структуры безопасного программирования, используемые для улучшения робастности, включая вставляемый компилятором код для проверок диапазонов и индексов массивов, подпрограммы обработки ошибок и исключений, проверки границ и допустимости, средства управления очередями,

отметки времени [1].

Ниже приведены некоторые примеры использования отключенного кода:

- отладочный код, используемый во время тестирования или поиска неисправностей в лаборатории. При установке в воздушное судно этот отладочный код отключается.
- функции обслуживания, используемые на земле уполномоченным техническим специалистом. Во время эксплуатации воздушного судна эти функции обслуживания деактивируются.
- программное обеспечение с выбором опций, которое используется вместо аппаратных выводов для выбора заранее одобренной конфигурации ПО в соответствии с потребностями конкретного эксплуатанта. Не выбранное ПО отключается.
- подмножество операционной системы, разработанное для выполнения целей DO-178C и удовлетворения потребностей нескольких пользователей. Неиспользуемые возможности деактивируются.
- система управления полетом, спроектированная для приема входных данных от нескольких датчиков, которые могут выбираться в зависимости от конфигурации воздушного судна. Код неиспользуемого датчика отключается.
- контроллер двигателя, разработанный для конфигурации с одним или двумя двигателями. Либо код для одного двигателя, либо код для двух двигателей отключается.
- выбранные функции из библиотеки, рассчитанной на потребности нескольких команд разработки программного обеспечения. Неиспользуемые библиотечные функции деактивируются.
- опции компилятора, например `#ifdef`, добавленные для применения одного и того же кода в нескольких местах, например для нескольких каналов или нескольких платформ. При компиляции такой код удаляется из исполняемого объектного кода и, следовательно, считается отключенным.

Этот список можно продолжать очень долго. Отключенный код обеспечивает настраиваемость программного обеспечения для нужд разных пользователей. Как видно из определения отключенного кода и приведенных примеров, такой код делится на две базовые категории: 1) код, который никогда не будет использоваться в полете, и 2) код, который может использоваться в какой-то момент времени в зависимости от конфигурации системы. В таблице 17.1 кратко показаны эти две категории и некоторые соображения, относящиеся к каждой из них.

Таблица 17.1 Обзор категорий отключенного кода

Категория	Описание и соображения
Категория один	Код, который никогда не будет активирован в ходе эксплуатации воздушного судна. Например, отладочный код. Отключенный код должен быть отключен во время эксплуатации воздушного судна. Необходимо доказать, что механизм деактивации действительно работает, то есть для него нужны требования и тесты. Подход к деактивации должен поддерживать уровень ПО активного программного обеспечения. Сам отключенный код обычно рассматривается с точки зрения сертификации как ПО уровня E. Однако все равно рекомендуется иметь требования и некоторый уровень тестирования отключенного кода, чтобы убедиться, что он выполняет задуманную функцию.

Категория	Описание и соображения
Категория два	Код, который может использоваться в одних конфигурациях, но не использоваться в других. Отключенный код должен быть связан с требованиями так же, как и любое другое бортовое ПО. Поскольку этот код предназначен для будущего использования, для него должна быть обеспечена гарантия на надлежащем уровне ПО. Иными словами, отключенное ПО должно удовлетворять применимым целям DO-178С для своего уровня ПО. Подход к деактивации, гарантирующий правильную деактивацию и конфигурирование, должен быть включен в требования и проект. Подход к деактивации должен соответствовать требуемому уровню гарантии разработки и должен быть протестирован.

Однажды мне довелось столкнуться с компанией, которая в конце проекта попыталась объявить свой мертвый код отключенным. Но между ними есть большая разница: отключенный код планируется и проектируется, а мертвый код – нет.

На рисунке 17.2 кратко показаны процессы, которые следует учитывать в отношении отключенного кода, а также ссылки на соответствующие разделы DO-178С. Процессы планирования, разработки и верификации описаны в следующих подразделах.

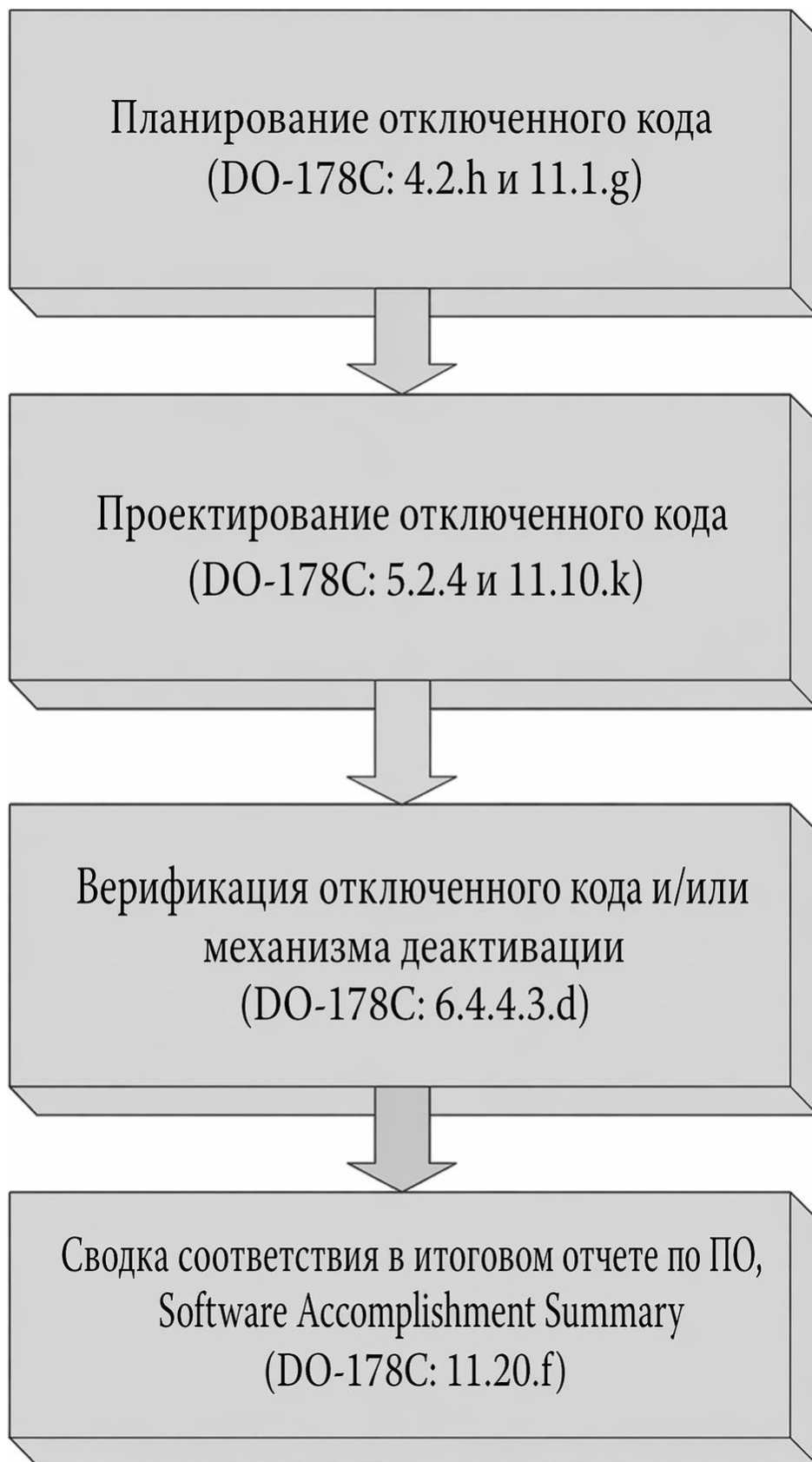


Рисунок 17.2 Сводка процесса для отключенного кода.

17.3.1 Планирование

Одно из главных отличий постороннего или мертвого кода от отключенного кода состоит в том, что отключенный код планируется. Мне еще не встречались люди, которые бы планировали добавить мертвый код, они планируют только то, как с ним разбираться, если

он появится, но не его реализацию. Отключенный код следует описывать в плане сертификации ПО (Plan for Software Aspects of Certification, PSAC), а также в плане разработки ПО (Software Development Plan, SDP) и плане верификации ПО (Software Verification Plan, SVP). Подход к отключенному коду обычно включается в план сертификации ПО как дополнительное соображение. В плане сертификации ПО следует объяснить, какие типы отключенного кода планируются, к какой категории относится отключенный код, то есть *категория один* или *категория два*, какие механизмы деактивации будут использоваться, а также как будет организована разработка и верификация самого отключенного кода и средств его деактивации. План разработки ПО раскрывает более подробную информацию о подходе к разработке отключенного кода. Важно дать разработчикам четкие указания о том, как обращаться с отключенным кодом на этапах требований, проектирования и кодирования. План верификации ПО содержит детали подхода к верификации как самого отключенного кода, особенно для кода *категории два*, так и механизмов деактивации.

17.3.2 Разработка

Как следует из определения отключенного кода в DO-178C, такой код определяется требованиями и документируется в проекте. В разделе 5.2.4 DO-178C приведено руководство, названное мероприятиями, по проектированию отключенного кода [1].

Во-первых, механизм деактивации должен быть спроектирован и реализован так, чтобы отключенный код не оказывал неблагоприятного влияния на активный код. Существует множество способов деактивации кода. Независимо от того, какой именно механизм используется, он должен учитываться в процессе оценки безопасности, а также должен быть верифицирован на предмет того, что действительно деактивирует нужный код. Ниже приведены некоторые примеры механизмов деактивации:

- использование модуля индивидуализации воздушного судна для выбора соответствующей конфигурации для конкретного борта. Такой модуль может быть как физическим модулем с переключателями, так и файлом конфигурации.
- выбор выводов для определения конфигурации. Если используется этот подход, рекомендуется задействовать два или более вывода, потому что один вывод слишком легко может замкнуться или разомкнуться. Аналогично обычно используют несколько сигналов, например одного только сигнала «судно на колесах» (weight-on-wheels) недостаточно, потому что его можно неверно интерпретировать. Возможные отказы должны рассматриваться командой по безопасности.
- использование умного компилятора, который не компилирует библиотечные функции, не включенные в код, тем самым удаляя отключенный код из исполняемого образа.
- использование опции компилятора, такой как `#ifdef`, для удаления кода из исполняе-

мого образа.

Во-вторых, должны существовать свидетельства того, что отключенный код не активен в средах, где он не должен использоваться или где он не одобрен. Для этого требуется эффективное управление конфигурацией отключенного кода и его применения. Управление конфигурацией должно рассматриваться как на уровне ПО, так и на уровне системы и воздушного судна. Например, если в ходе разработки используется библиотека функций, должен существовать процесс, обеспечивающий применение только одобренных библиотечных функций. Если не проявить осторожность, будущие разработчики могут просто предположить, что одобрена вся библиотека целиком. Один из способов справиться с этим состоит в выпуске только одобренных функций, чтобы неутвержденные функции были недоступны.

В-третьих, сам отключенный код должен соответствовать применимым целям DO-178C. Применимые цели зависят от категории кода. Для отключенного кода *категории один* обычно приемлем уровень E, поскольку такое программное обеспечение не будет использоваться в ходе полетной эксплуатации. Однако если отключенный код будет применяться для принятия важных решений по техническому обслуживанию или для выполнения важных операций обслуживания, может потребоваться его разработка на более высоком уровне. К отключенному коду *категории два* в ходе разработки следует относиться так же, как к активному коду, поскольку предполагается его использование в некоторых конфигурациях.

17.3.3 Верификация

Как сам отключенный код, так и механизм его деактивации должны быть верифицированы в соответствии с надлежащим уровнем ПО DO-178C. Если компилятор не является квалифицированным инструментом, а в качестве механизма деактивации выбраны *умная компиляция* или параметры компилятора, важно проанализировать выходные данные компилятора, чтобы убедиться, что отключенный код действительно удален из исполняемого образа.

Это подводит к довольно распространенной и спорной проблеме, связанной с отключенным кодом. Иногда бывает непросто отнести отключенный код к *категории один* или *категории два*. Цель проекта может состоять в том, чтобы иметь отключенный код *категории два* для удовлетворения потребностей нескольких заказчиков. Однако фактическая реализация может в итоге оказаться иной из-за графика работ или изменения направления проекта. Это может привести к тому, что часть кода *категории два* будет

переведена в *кате́горию один*. Такой шаг необходимо тщательно оценивать и обсуждать с сертифицирующим органом. Если применяется именно такой подход, должны существовать конкретные ограничения, не позволяющие пользователям и будущим разработчикам считать, что отключенный код одобрен для будущего использования.

Ссылки

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).

Глава 18. Загружаемое в полевых условиях программное обеспечение

Сокращения

Сокращение	Расшифровка
CD	Компакт-диск
DAL	Уровень гарантии разработки
DVD	Цифровой видеодиск
FAA	Федеральное управление гражданской авиации США
FADEC	Цифровая система управления двигателем с полной ответственностью
FLS	Загружаемое в полевых условиях программное обеспечение
LAN	Локальная вычислительная сеть
LRM	Линейно-заменяемый модуль
LRU	Линейно-заменяемый блок
PMAT	Портативный многоцелевой терминал доступа
PSAC	План сертификации ПО
SCI	Указатель конфигурации ПО
SCMP	План управления конфигурацией ПО
SDP	План разработки ПО
SQA	Гарантия качества ПО
SQAP	План гарантии качества ПО
SVP	План верификации ПО

18.1 Введение

В самый первый день моей работы в штаб-квартире Федерального управления гражданской авиации США (Federal Aviation Administration, FAA), в роли национального руководителя программного направления, меня познакомили с проблемами, связанными с загружаемым в полевых условиях программным обеспечением (field-loadable software, FLS). Иногда я думаю, не было ли это своего рода злобным посвящением со стороны коллег.

Вероятно, они стояли по другую сторону офисной перегородки и покатывались со смеху...

Начиналось все как вполне обычный первый рабочий день. После переезда из Канзаса в район Вашингтона, округ Колумбия, мне пришлось разобраться с поездом Virginia Railway Express, найти нужный этаж в здании штаб-квартиры Федерального управления гражданской авиации США, получить пропуск, заполнить целую прорву бумаг, познакомиться со

всеми в подразделении авионики и добраться до своего рабочего места. Пока я возилась с телефоном, настраивая голосовую почту, он зазвонил. Я подумала, что это административный помощник сообщает мне о чем-то, что мы забыли обсудить. Но это оказался юрист, очень громкая и очень злая женщина из одной из вашингтонских авиационных организаций из так называемого буквенного супа. Она немедленно начала отчитывать меня за недостаточную отзывчивость Федерального управления гражданской авиации США по вопросу необходимости руководящих указаний по одобрению изготовителя частей (Parts Manufacturer Approval, PMA) для загружаемого в полевых условиях программного обеспечения и так далее, и так далее, и так далее. Я все время напоминала себе: «Мы уже не в Канзасе, Тото».

18.2 Что такое загружаемое в полевых условиях программное обеспечение?

DO-178C (КТ-178С) описывает загружаемое в полевых условиях программное обеспечение как «ПО, которое может быть загружено без снятия системы или оборудования с места установки» [1]. Оно загружается в оборудование самолета или двигателя через некоторый порт данных, без необходимости вскрывать блок. Федеральное управление гражданской авиации США также относит к такому ПО ПО, загружаемое через порт данных в квалифицированных лабораториях, например на ремонтных станциях или в сервисных центрах [2]. По существу, загружаемое в полевых условиях ПО противопоставляется ПО, загружаемому на заводе, когда для загрузки ПО в линейно-заменяемый блок (line-replaceable unit, LRU) или линейно-заменяемый модуль (line-replaceable module, LRM) может потребоваться нарушение пломбы блока, например перепрошивка.

Носители, используемые для загрузки такого программного обеспечения, со временем меняются. В начале 1990-х использовались дискеты 3,5 дюйма. Например, на первых Boeing 777 возили с собой целые папки, набитые гибкими дисками. Затем технология двинулась дальше: появились компакт-диски (CD), цифровые видеодиски (DVD), флэш-накопители, устройства массового хранения и даже локальные вычислительные сети (LAN).

Если приняты надлежащие меры безопасности и программное обеспечение одобрено сертифицирующими органами, почти любое оборудование может поддерживать полевую загрузку: от систем управления полетом и управления двигателем до навигационных и связанных систем, систем управления полетом и систем предупреждения столкновений в воздухе. Выстроить процессы и разработать системы так, чтобы они были пригодны для полевой загрузки, вовсе не тривиальная задача. Приходится учитывать множество нормативных требований, например правила маркировки составных частей и требования к одобренным ремонтным станциям, причем на нескольких уровнях: уровне оборудования, уровне воздушного судна, уровне летной эксплуатации. Тем не менее такое ПО сегодня очень широко распространено в авиационной отрасли.

Следует отметить, что хотя аэронавигационные данные, например навигационные базы или базы рельефа, тоже загружаются в полевых условиях, с ними обращаются не так, как с загружаемым в полевых условиях программным обеспечением. В общем случае для аэронавигационных данных применяется руководство DO-200А, тогда как для такого ПО применяются положения DO-178С. Конкретные сведения по аэронавигационным данным приведены в главе 23.

18.3 Преимущества загружаемого в полевых условиях программного обеспечения

Загружаемое в полевых условиях программное обеспечение позволяет сделать процессы технического обслуживания более эффективными и лучше соответствующими потребностям мирового парка воздушных судов. Вместо того чтобы пересылать линейно-заменяемые блоки или линейно-заменяемые модули, снимать оборудование с самолета для модернизации и держать большой запас дорогого оборудования, можно распространять ПО и загружать его прямо на борту. Это сокращает время простоя воздушного судна и помогает удовлетворять потребности авиакомпаний и пассажиров.

18.4 Трудности загружаемого в полевых условиях программного обеспечения

Как и почти все в жизни, преимущества не достаются без борьбы. При внедрении процессов, последствия которых затрагивают жизни людей по всему миру, необходимо решить несколько сложных задач. Кроме того, в управлении этими трудностями участвует несколько заинтересованных сторон, включая разработчиков программного обеспечения и оборудования, производителей самолетов или двигателей, авиакомпании и сертифицирующие органы. Некоторые из таких трудностей перечислены ниже:

- Проектирование системы таким образом, чтобы она была безопасной, что включает защиту от несанкционированных изменений, выполнение проверок целостности и так далее.
- Управление конфигурацией на нескольких уровнях, то есть на уровне ПО, оборудования и самолета или двигателя, чтобы гарантировать, что: (1) утвержденное ПО установлено в утвержденное оборудование и (2) утвержденное оборудование установлено на сертифицированный самолет или двигатель.
- Применение нормативных требований, ориентированных на оборудование, к программному обеспечению, например требований по маркировке составных частей. Большинство норм написано для самолета или двигателя и установленных на них систем. Поскольку загружаемое в полевых условиях ПО становится самостоятельной частью, подобные требования приходится интерпретировать так, чтобы применять их

на уровне ПО.

- Обеспечение того, чтобы все загружаемое программное обеспечение было одобрено сертифицирующим органом и чтобы на самолет или двигатель не устанавливались неодобренные составные части.
- Обеспечение отсутствия проблем информационной безопасности, например вирусов, злоумышленников и тому подобного, особенно когда программное обеспечение передается или загружается по сети. Также требуется удостовериться, что ПО загружено полностью и без повреждения. Возможность загрузки должна быть отключена во время полета.
- Управление совместимостью с другим оборудованием при обновлении программного обеспечения. Обновления могут повлиять на оборудование, взаимодействующее с ПО, поэтому оборудование, взаимодействующее с этим ПО, также должно оцениваться.

18.5 Разработка и загрузка загружаемого в полевых условиях программного обеспечения

К счастью, после двух десятилетий работы с таким программным обеспечением многие из этих трудностей были преодолены. Сегодня такое ПО используется повсеместно. В этом разделе рассматриваются четыре направления, которые необходимо учитывать при работе с загружаемым в полевых условиях ПО: (1) разработка систем, пригодных для полевой загрузки, (2) разработка самого ПО, (3) загрузка ПО и (4) модификация ПО.

18.5.1 Разработка системы, пригодной для полевой загрузки

Проектирование такого программного обеспечения связано не только с самим ПО. Система как таковая тоже должна быть спроектирована так, чтобы обеспечивать безопасную загрузку ПО в полевых условиях. В разделе 2.5.5 DO-178C указано несколько аспектов, которые следует учитывать при разработке системы для безопасной загрузки ПО в полевых условиях, в том числе следующие [1]:

- *Защита от непреднамеренной загрузки*, например от загрузки во время полета. Такая защита может обеспечиваться аппаратными средствами, программными средствами или их сочетанием. В анализе безопасности должно быть рассмотрено влияние отказа механизма обнаружения. Если иное не обосновано в анализе безопасности, механизм обнаружения обычно разрабатывается по наивысшему уровню гарантии разработки (Development Assurance Level, DAL) среди загружаемого программного обеспечения.
- *Обнаружение неудачных, частичных или поврежденных загрузок*. Согласно разделу 2.5.5.b DO-178C, если при обнаружении частичной, поврежденной или неподходящей загрузки программного обеспечения система переводится в безопасный режим или

режим по умолчанию, то «для каждого обособленного компонента системы должны быть заданы требования, связанные с безопасностью, для восстановления до этого режима и работы в нем» [1].

- *Определение неподходящих, то есть некорректных, загрузок программного обеспечения*, например случаев, когда по ошибке загружается не тот файл или ПО вообще не загружается.
- *Целостность механизма отображения*, используемого для проверки конфигурации воздушного судна, то есть для подтверждения того, что загружено правильное программное обеспечение. Также необходимо учитывать потерю или повреждение возможности отображать обозначение ПО. Это рассматривается только в том случае, если для проверки загрузки ПО используется бортовая система отображения.

Для проработки этих деталей требуются системные требования и системный проект, а также участие специалистов по системной безопасности. Желательно подключать специалистов по безопасности как можно раньше. Недавно мне довелось обнаружить недостаток в стратегии одной компании по полевой загрузке на уровне системы во время анализа плана сертификации ПО (Plan for Software Aspects of Certification, PSAC). Этот план готовится после того, как определена архитектура системы и выполнен первоначальный анализ безопасности. Это обнаружение привело к переработке системы на поздней стадии проекта, хотя этого можно было избежать, если бы специалисты по безопасности были вовлечены раньше.

Еще одно системное соображение связано с подходом к идентификационной маркировке такого программного обеспечения. Некоторые производители обновляют обозначение аппаратного изделия всякий раз, когда загружается такое ПО. Другие ведут обозначения аппаратуры и ПО отдельно. Допустим любой из этих подходов, главное, чтобы он был определен как часть системного проекта.

18.5.2 Разработка загружаемого в полевых условиях программного обеспечения

Как и любое другое программное обеспечение, загружаемое в полевых условиях ПО разрабатывается в соответствии с DO-178C и/или применимыми дополнениями. Кроме того, само такое ПО должно быть спроектировано так, чтобы поддерживать полевую загрузку. Обычно это означает реализацию проверки целостности, например циклического избыточного кода (cyclic redundancy check, CRC), проектирование ПО с учетом перечисленных выше вопросов и разработку приложения для полевой загрузки, нередко отдельного приложения. Многие заявители проектируют свои системы в соответствии с ARINC 615, который содержит как «общие, так и специальные рекомендации по проектированию оборудования для загрузки программных данных для всех типов воздушных судов» [3]. Кроме того, для терминала

технического обслуживания может применяться ARINC 644. В ARINC 644 приведены рекомендации по разработке *портативного многоцелевого терминала доступа*, используемого авиакомпаниями [4].

Планирование является важной частью любого проекта по DO-178С. Для такого программного обеспечения это особенно верно. Планы описывают аспекты разработки, верификации, управления конфигурацией, гарантии качества и взаимодействия с сертифицирующим органом применительно к такому ПО. План сертификации ПО описывает подход к такому ПО, обеспечивает учет ранее отмеченных вопросов, описывает используемую проверку целостности и ее достаточность для данного уровня ПО, то есть точность, а также излагает детали управления конфигурацией при разработке и верификации такого ПО. План разработки ПО (Software Development Plan, SDP) развивает положения Плана сертификации ПО и предоставляет подробности для команды разработки такого ПО; в нем объясняются механизмы защиты, проверки целостности и любые специальные стандарты, которые применяются, например стандарт ARINC 615. План верификации ПО (Software Verification Plan, SVP) содержит подробности, необходимые для того, чтобы такое ПО, проверки целостности и защитные механизмы были верифицированы и протестированы. План управления конфигурацией ПО (Software Configuration Management Plan, SCMP) описывает процесс управления конфигурацией для такого ПО. На этапе разработки управление конфигурацией, вероятно, будет таким же, как и для ПО, не предназначенного для полевой загрузки, однако потребуются объяснить, кто отвечает за управление конфигурацией после завершения разработки. Обычно это также описывается и в Плане сертификации ПО, и в Плане управления конфигурацией ПО.

После того как планы разработаны, им следует следовать, как и в случае любого другого бортового программного обеспечения.

18.5.3 Загрузка загружаемого в полевых условиях программного обеспечения

После того как такое программное обеспечение разработано и одобрено сертифицирующим органом, его можно загружать. Большинство заявителей используют проверку целостности, например циклический избыточный код, чтобы убедиться, что ПО не было повреждено в процессе загрузки. Достаточность проверки целостности должна быть подтверждена еще на этапе разработки. Точность обычно определяется алгоритмом проверки целостности и размером защищаемого файла или файлов. Для больших файлов может потребоваться использовать проверки целостности большей разрядности, например 32 бита вместо 8 или 16, либо разделять данные на более мелкие пакеты и применять дополнительные проверки целостности. Вероятность ошибки должна соответствовать уровню загружаемого ПО.

Если надежная проверка целостности не используется, оборудование, применяемое для за-

грузки такого программного обеспечения, может потребовать определенной квалификации. Например, ПО загрузчика данных может потребоваться разрабатывать в соответствии с DO-178С или DO-330, а загрузка ПО может быть разрешена только с использованием одобренного загрузчика данных. Сегодня такой подход применяется редко.

Согласно FAA Order 8110.49, процесс загрузки такого программного обеспечения также должен обеспечивать следующее [2]:

- Загружаемое программное обеспечение одобрено сертифицирующим органом.
- Процедура загрузки одобрена и соблюдается. Как отмечалось в главе 10, процедуры загрузки обычно включаются в Указатель конфигурации ПО (Software Configuration Index, SCI) или на него дается ссылка. Если они не относятся к ответственности команды ПО, они могут быть включены в документацию системного уровня или уровня воздушного судна. В этом случае Указатель конфигурации ПО должен это пояснять.
- Программное обеспечение одобрено для того оборудования, на которое оно загружается, то есть представляет собой одобренную конфигурацию аппаратуры и ПО.
- Программное обеспечение одобрено для того воздушного судна или двигателя, на которое оно загружается, то есть представляет собой одобренную конфигурацию воздушного судна или двигателя. Это включает обеспечение того, что любые резервированные части на воздушном судне или двигателе сконфигурированы надлежащим образом. Например, если имеется два цифровых блока управления двигателем с полной ответственностью (Full Authority Digital Engine Control, FADEC), то при загрузке обновленного ПО может потребоваться загрузить его в оба блока, если только конфигурация воздушного судна или двигателя не допускает, чтобы один такой блок имел новую версию ПО, а другой – старую.
- Программное обеспечение загружено полностью и без ошибок. Обычно это проверяется подтверждением того, что циклический избыточный код совпадает со значением, указанным в инструкциях по загрузке.
- После загрузки подтверждено обозначение программного обеспечения, включая номер версии, если он применим.
- Изменение конфигурации задокументировано в записях о конфигурации воздушного судна или двигателя, а также в любых других применимых записях по техническому обслуживанию.

18.5.4 Модификация загружаемого в полевых условиях программного обеспечения

Одной из причин распространения такого программного обеспечения является то, что ПО можно модифицировать и загружать без возврата оборудования изготовителю.

Когда такое программное обеспечение модифицируется, оно должно проходить процесс анализа воздействия изменений, как и любое другое бортовое ПО. Все измененные и затронутые данные и код повторно верифицируются. Сведения об анализе воздействия изменений приведены в главе 10. ПО должно быть повторно верифицировано для целевой аппаратуры и конфигурации воздушного судна или двигателя, на которых предполагается установка. Кроме того, до установки модифицированное ПО должно быть одобрено сертифицирующим органом для соответствующего оборудования и конфигурации воздушного судна или двигателя.

18.6 Заключение

Загружаемое в полевых условиях программное обеспечение стало обычным явлением в современной авиационной отрасли. Однако проектирование системы и ПО, вопросы загрузки и управление конфигурацией по-прежнему должны тщательно контролироваться, особенно по мере выхода на рынок новых участников и развития подходов к загрузке.

Ссылки

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
2. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Washington, DC: Federal Aviation Administration, Change 1, September 2011).
3. Aeronautical Radio, Inc., *Software data loader using ethernet interface*, ARINC REPORT 615A-3 (Annapolis, MD: Airlines Electronic Engineering Committee, June 2007).
4. Aeronautical Radio, Inc., *Portable multi-purpose access terminal (PMAT)*, ARINC REPORT 644A (Annapolis, MD: Airlines Electronic Engineering Committee, August 1996).

Глава 19. Модифицируемое пользователем ПО

Сокращения

Сокращение	Расшифровка
ACARS	Система адресации и передачи сообщений с борта воздушного судна
ACMS	Система мониторинга состояния воздушного судна
AMI	Информация, модифицируемая авиакомпанией
EASA	Агентство Европейского союза по авиационной безопасности
FAA	Федеральное управление гражданской авиации США
PSAC	План сертификации ПО
SCI	Указатель конфигурации ПО
SDP	План разработки ПО
SQA	Гарантия качества ПО
SVP	План верификации ПО
UMS	Модифицируемое пользователем ПО

19.1 Введение

В этой главе определяется модифицируемое пользователем ПО (User-Modifiable Software, UMS) и приводится несколько примеров такого программного обеспечения. Затем глава объясняет, какие сертификационные указания следует учитывать при проектировании бортовой системы, содержащей модифицируемое пользователем ПО. Последний раздел сосредоточен на пользователе, например авиакомпании, который будет изменять и сопровождать ПО после сертификации системы.

19.2 Что такое модифицируемое пользователем ПО?

Модифицируемое пользователем ПО – это «ПО, которое можно модифицировать без согласования с сертифицирующим органом, самолетостроителем или поставщиком оборудования, если модификация происходит в пределах, установленных в исходном проекте сертификации» [1]. В документе ARINC 667-1 под названием *Guidance for management of field loadable software* модифицируемое пользователем ПО описывается так: «ПО, предназначенное для модификации эксплуатантом воздушного судна без рассмотрения со стороны сертифицирующего органа, держателя ТС [сертификата типа] или STC [дополнительного сертификата типа], либо изготовителя оборудования. Модификации могут включать изменения данных, исполняемого кода или того и другого» [2].[*] Такое ПО существует в разных формах, включая исполняемый объектный код, специфические для конкретного воздушного

судна пользовательские настройки параметров или базы данных [2]. Пользователем обычно является эксплуатант воздушного судна или авиакомпания. Однако возможны ситуации, когда пользователем выступает заказчик оборудования, например изготовитель воздушного судна, устанавливающий на свое воздушное судно изделие, одобренное по Техническому стандартному приказу (Technical Standard Order, TSO), возможно как готовое коммерческое ПО (Commercial-Off-The-Shelf software, COTS).

Модифицируемое пользователем ПО обычно проектируется и первоначально одобряется как часть бортовой системы, содержащей как немодифицируемое программное обеспечение, например ПО управления полетом, мониторинга двигателя или навигации, так и модифицируемое ПО, например контрольные перечни, модифицируемые авиакомпанией. Немодифицируемое ПО не проектируется и не предназначается для изменения пользователем, и именно оно было основной темой этой книги. Такое ПО, напротив, специально проектируется так, чтобы пользователь мог изменять его с использованием одобренных процедур. Немодифицируемое ПО должно быть защищено от любых изменений со стороны пользователя. Как будет обсуждаться далее в этой главе, для такой защиты существует несколько вариантов.

Во время первоначальной сертификации система проектируется так, чтобы допускать пользовательскую модификацию и одновременно защищать немодифицируемое программное обеспечение от пользовательских изменений. Кроме того, процедуры для модифицируемого пользователем ПО одобряются вместе с первоначальным одобрением системы. Поскольку модифицируемое пользователем ПО находится вне контроля изготовителей оборудования и воздушного судна, при первоначальной сертификации его обычно рассматривают как ПО уровня E. Некоторые изготовители могут относить его к уровню D, но это делается скорее в целях качества, а не ради сертификации.

И DO-178C (KT-178C), и глава 7 приказа FAA 8110.49 рассматривают модифицируемое пользователем ПО. Эти документы сосредоточены на бортовом программном обеспечении, которое одобряется как часть сертификации типа. В них подчеркивается необходимость спроектировать систему так, чтобы она была модифицируемой, и защитить немодифицируемое ПО от модифицируемого пользователем ПО. Оба документа также объясняют, что после одобрения в качестве модифицируемого пользователем ПО такое ПО больше не требует участия сертифицирующего органа.

Однако одобрение со стороны органа эксплуатационного надзора все еще может требоваться. Сертификация типа и эксплуатационное одобрение воздушного судна обычно выдаются разными подразделениями сертифицирующего органа. Например, в США офисы FAA по сертификации авиационной техники выдают сертификат типа на воздушное судно,

а подразделения Flight Standards выдают эксплуатационное одобрение. Поэтому с точки зрения пользователя могут существовать два класса такого программного обеспечения: 1) требующее эксплуатационного одобрения и 2) не требующее эксплуатационного одобрения. Первую категорию некоторые организации обозначают терминами *airline certifiable software* или *user certifiable software*, то есть как ПО, для которого пользователь или авиакомпания сами получают одобрение органа эксплуатационного надзора [2].[†]

19.3 Примеры модифицируемого пользователем ПО

Тип и объем модифицируемого пользователем ПО различаются от воздушного судна к воздушному судну и от авиакомпании к авиакомпании. Один мой недавний студент, работающий в крупной международной авиакомпании, рассказывал, что у одного из типов воздушных судов, которыми он управляет, имеется около 130 модифицируемых компонентов. Ниже приведены лишь несколько примеров такого программного обеспечения.

Пример 1: Система адресации и передачи сообщений с борта воздушного судна (Aircraft Communications Addressing and Reporting System, ACARS). Эта система обеспечивает возможность обмена сообщениями между воздушным судном и наземными станциями. Пользователи могут оптимизировать систему под нужды конкретной авиакомпании. Среди ее функций – выявление нештатных условий полета, поддержка ремонта и планирования технического обслуживания, обмен сообщениями электронной почты между экипажем и органами управления воздушным движением, метеосводки и отчеты по двигателю.

Пример 2: Система мониторинга состояния воздушного судна (Airplane Condition Monitoring System, ACMS). Эта система позволяет пользователю собирать данные, необходимые для принятия решений по техническому обслуживанию. Пользователи могут программировать ее на регистрацию конкретных данных для последующего анализа. Система получает данные от бортовой системы, но не передает данные в немодифицируемое программное обеспечение.

Пример 3: Контрольные перечни, специфичные для авиакомпании. Некоторые авиакомпании имеют программируемые контрольные перечни для своих конкретных операций. Такие перечни могут требовать эксплуатационного одобрения со стороны регулирующего органа, но не одобряются как часть сертификации типа.

Пример 4: Модифицируемая панель автоматических выключателей. Модифицируемая панель автоматических выключателей позволяет добавлять после сертификации типа оборудование, не являющееся обязательным, например кофеварку, систему развлечений в полете или стереосистему. Модифицируемое пользователем ПО используется для задания пределов срабатывания автоматических выключателей на основе пользовательского ввода.

Пример 5: Система табличек и световой индикации в салоне. Таблички и система световой индикации различаются от авиакомпании к авиакомпании. Поэтому некоторые авиакомпании программируют их с использованием модифицируемого пользователем ПО. Такое модифицируемое пользователем ПО требует эксплуатационного одобрения, но не одобряется как часть сертификата типа воздушного судна.

Пример 6: Информация, модифицируемая авиакомпанией (Airline-Modifiable Information, AMI). Это файл данных, позволяющий пользователям задавать предпочтения для данных управления салоном, регистрации, форматов отчетов и услуг, предоставляемых в различных зонах размещения пассажиров, например в первом классе, бизнес-классе и эконом-классе.

19.4 Проектирование системы для модифицируемого пользователем ПО

Системы с модифицируемым пользователем ПО должны быть спроектированы так, чтобы их можно было модифицировать. DO-178C (разделы 2.5.2, 5.2.3), приказ FAA 8110.49 (глава 7) и сертификационный меморандум EASA CM-SWCEH-002 (глава 9) дают представление о том, как проектировать систему с учетом модифицируемого пользователем ПО [1,3,4]. В этом разделе кратко суммируется соответствующий процесс в виде *рекомендаций для разработчика*, применимых на этапе первоначальной сертификации пользовательской системы. Следующий раздел, раздел 19.5, затем приводит *рекомендации* для тех, кто будет модифицировать программное обеспечение после сертификации.

Рекомендация разработчику 1: учитывать аспекты модифицируемого пользователем ПО при сертификации типа и разработке системы. Предусматриваемое наличие модифицируемого пользователем ПО следует планировать, в частности в Плане сертификации ПО (PSAC), Плане разработки ПО (SDP) и, возможно, в Плане верификации ПО (SVP), а также согласовывать с сертифицирующим органом. В Плане сертификации ПО (PSAC) модифицируемое пользователем ПО обычно описывается как *дополнительный аспект, требующий учета*. Требования и проектные решения должны быть подготовлены так, чтобы поддерживать модифицируемое пользователем ПО и защищать немодифицируемое программное обеспечение. Такие требования также должны быть верифицированы.

Само модифицируемое пользователем ПО не одобряется как часть сертификата типа. Как отмечено в DO-178C, раздел 7.2.2.b: «Модифицируемое пользователем ПО не включается в базовую версию программного продукта, за исключением связанных с ним компонентов защиты и компонентов, задающих границы. Следовательно, в модифицируемое пользователем ПО могут вноситься изменения без изменения идентификации конфигурации базовой версии программного продукта» [1]. По сути, система, одобренная сертифицирующим органом, будет способна поддерживать модифицируемое пользователем ПО, но не будет

включать само это программное обеспечение.

Рекомендация разработчику 2: убедиться, что модифицируемое пользователем ПО не может неблагоприятно повлиять на данные, связанные с безопасностью. Само модифицируемое пользователем ПО и изменения в нем не должны влиять ни на один из следующих видов данных, связанных с безопасностью: запасы безопасности, эксплуатационные возможности воздушного судна или двигателя, рабочая нагрузка летного экипажа, немодифицируемые компоненты, механизмы защиты, программные границы, например заранее верифицированные диапазоны данных [3]. Эти аспекты следует рассматривать в рамках процесса оценки безопасности системы. Если такое программное обеспечение или изменение в нем может повлиять на любые данные, связанные с безопасностью, такое ПО не должно классифицироваться как модифицируемое пользователем ПО.

Рекомендация разработчику 3: привлекать надлежащих заинтересованных лиц. Поскольку модифицируемое пользователем ПО пересекает границы летной годности и эксплуатации, то есть границы между проектированием воздушного судна и системы и эксплуатацией пользователем, важно вовлекать всех необходимых участников, включая сертифицирующий орган, орган эксплуатационного одобрения, изготовителя воздушного судна, системного разработчика, разработчика ПО, специалистов по безопасности и пользователей. Координация с этими сторонами на ранних этапах и на протяжении всей разработки критически важна для успеха.

Рекомендация разработчику 4: предусмотреть меры защиты немодифицируемого программного обеспечения. В разделе 5.2.3.a DO-178C подчеркивается, что «немодифицируемый компонент должен быть защищен от модифицируемого компонента» [1].[*] Защита должна быть робастной как во время эксплуатации модифицируемого пользователем ПО, так и во время его модификации. Защита может реализовываться аппаратурой, например отдельными процессорами, ПО, например обособлением ПО, инструментами, например пользовательским интерфейсом, ограничивающим допустимый ввод, либо сочетанием аппаратуры, ПО и инструментов [1]. Одним из примеров защиты является функция мониторинга, встроенная в основное ПО и предотвращающая любые изменения немодифицируемого ПО. Монитор и немодифицируемое ПО недоступны пользователю [2]. Уровень гарантии разработки защитного механизма должен совпадать с наиболее тяжелым отказным состоянием в системе, на которое может повлиять модифицируемое пользователем ПО [3]. Например, если бортовое немодифицируемое ПО имеет уровень А и для его защиты используется обособление ПО, то механизм обособления должен разрабатываться по уровню А. Сведения об обособлении приведены в главе 21.

Рекомендация разработчику 5: оценивать инструменты, используемые для обеспечения за-

щиты немодифицируемого программного обеспечения. Большинство систем, поддерживающих модифицируемое пользователем ПО, предоставляют пользователям инструмент или набор инструментов для его модификации. Необходимо учитывать влияние такого инструмента на немодифицируемое ПО. Во время первоначальной разработки системы следует рассматривать использование инструмента, управление им, его проект, функциональность, то есть выполняемые модификации такого ПО, и сопровождение. Инструмент должен быть спроектирован достаточно робастно, чтобы предотвращать ошибочный ввод со стороны пользователя. Раздел 7.6.b приказа 8110.49 идет еще дальше и утверждает: «ПО, образующее компонент инструмента и используемое в защитной функции, должно разрабатываться по уровню ПО наиболее тяжелого отказного состояния системы, определенного по результатам оценки безопасности системы» [3].

Существуют случаи, когда инструмент может потребовать квалификации. Раздел 7.6.c приказа 8110.49 поощряет квалификацию инструмента «и одобрение процедур его использования и сопровождения» [3]. К сожалению, указания относительно того, когда именно инструмент требует квалификации, несколько неясны. Раздел 12.2 DO-178C содержит руководящие указания о том, когда инструменты, используемые в процессах DO-178C, должны квалифицироваться и до какого уровня. Однако в случае модифицируемого пользователем ПО здесь остается неоднозначность, поскольку инструменты, обеспечивающие модифицируемое пользователем ПО, обычно не сокращают, не автоматизируют и не устраняют процесс по DO-178C. В целом я рекомендую следующее: если защитный механизм опирается только на инструмент, инструмент следует квалифицировать; наоборот, если влияние инструмента компенсируется архитектурой системы, например встроенным монитором, квалификация может не потребоваться. Даже если инструмент не требует квалификации, он все равно должен быть специфицирован, верифицирован и находиться под управлением конфигурации. Решение о том, квалифицировать инструмент или нет, должно согласовываться с сертифицирующим органом. Дополнительные сведения о квалификации инструментов приведены в главе 13.

Рекомендация разработчику 6: учитывать отображаемые данные. Приказ FAA 8110.49, раздел 7.3, поясняет, что любые данные, отображаемые летному экипажу и определяемые модифицируемым пользователем ПО, должны быть явно обозначены для экипажа как справочные либо должны иметь эксплуатационное одобрение. По сути, важно, чтобы пилоты понимали целостность наблюдаемых ими данных и могли правильно ими пользоваться. Такой тип данных необходимо тесно согласовывать со специалистами по летным испытаниям и человеческому фактору во время первоначальной сертификации воздушного судна.

Рекомендация разработчику 7: предотвращать непреднамеренные и несанкционированные изменения. Модифицируемое пользователем ПО должно изменяться только уполномочен-

ным персоналом на земле с использованием одобренной процедуры. Во время первоначальной сертификации необходимо убедиться, что одобренная процедура является единственным способом изменения модифицируемого пользователем ПО [3].

Рекомендация разработчику 8: информировать пользователей обо всем модифицируемом пользователем ПО. Держатель сертификата типа должен информировать пользователя воздушного судна обо всем таком программном обеспечении, имеющемся на воздушном судне. Кроме того, одобренные процедуры изменения модифицируемого пользователем ПО должны быть четко документированы и предоставлены пользователям. Такие процедуры должны быть воспроизводимыми и находиться под управлением конфигурации.

Рекомендация разработчику 9: определить ограничения и процедуры для пользователей. В ходе первоначальной разработки системы необходимо четко документировать ограничения и процедуры, чтобы пользователи обладали достаточными данными для внесения изменений в модифицируемое пользователем ПО без ущерба для безопасности воздушного судна. В разделе 11.16.j DO-178C указано, что Указатель конфигурации ПО (SCI) должен идентифицировать «процедуры, методы и инструменты внесения изменений в модифицируемое пользователем ПО, если таковые имеются» [1]. Такие процедуры должны быть предоставлены пользователям. Интересно, что представители авиакомпаний указывают, что именно в этой области они чаще всего сталкиваются с недостатками. Нередко авиакомпании получают от изготовителей воздушных судов или систем неясную или неполную документацию. Это вынуждает сотрудников авиакомпаний делать предположения, которые могут оказаться как верными, так и неверными. Это одна из причин, по которым пункт 11.16.j был добавлен в DO-178C.

Кроме того, любые изменения, влияющие на модифицируемое пользователем ПО, должны ясно доводиться до пользователей, например изменения в системе, поддерживающей модифицируемое пользователем ПО, обновления одобренных процедур или модификации инструмента, применяемого для изменения модифицируемого пользователем ПО. Эксплуатанты авиакомпаний указывают, что при обновлении системы, процедур или инструмента они редко получают актуализированные процедуры.

19.5 Модификация и сопровождение модифицируемого пользователем ПО

После того как программное обеспечение одобрено по сертификату типа как модифицируемое пользователем ПО, ответственность за управление этим ПО переходит к пользователю. Это включает управление средой, изменениями в модифицируемом пользователем ПО и общей конфигурацией воздушного судна. Доступные руководящие указания по модификации и сопровождению модифицируемого пользователем ПО весьма скудны, поэтому в этом разделе приводятся некоторые рекомендации для пользователей, которым приходится

управлять модифицируемым пользователем ПО. Поскольку типы такого ПО сильно различаются, эти рекомендации носят общий характер. Конкретные детали должны определяться совместно заинтересованными сторонами конкретного проекта.

Рекомендация пользователю 1: убедиться, что необходимые соглашения действуют. Пользователи должны убедиться, что с изготовителем воздушного судна и/или оборудования заключены необходимые соглашения, например договоры на техническую поддержку, чтобы получить все требуемые данные, инструменты и поддержку.

Рекомендация пользователю 2: управлять конфигурацией всего модифицируемого пользователем ПО на воздушном судне. Важно знать, какая версия программного обеспечения, как модифицируемого пользователем ПО, так и немодифицируемого ПО, установлена на каждом воздушном судне. Записи по техническому обслуживанию должны фиксировать, когда устанавливаются конкретные версии ПО или его обозначения.

Рекомендация пользователю 3: управлять средой каждого компонента модифицируемого пользователем ПО. Для каждого компонента модифицируемого пользователем ПО будет существовать своя среда, в которой выполняются изменение и модификация такого программного обеспечения. Инструменты, образующие эту среду, обычно предоставляются изготовителем оборудования или изготовителем воздушного судна. В качестве примеров можно назвать редактор ПО, компилятор, средство присвоения обозначения и средства формирования набора носителей [2]. Поддерживающие инструменты должны находиться под управлением конфигурации. Если изготовитель оборудования или воздушного судна предоставляет обновленные инструменты, влияние этих инструментов следует оценивать до их применения. Для понимания такого влияния может потребоваться координация с изготовителем оборудования или воздушного судна.

Рекомендация пользователю 4: модифицировать программное обеспечение с использованием предоставленных ограничений и процедур. Как отмечено в рекомендации разработчику N 9, изготовитель оборудования или воздушного судна должен определить ограничения и процедуры, которые пользователь обязан соблюдать. На эти ограничения и процедуры ссылаются данные сертификации типа, то есть они одобрены как часть сертификата типа. Следовательно, предполагается, что пользователь будет соблюдать эти процедуры. Если процедуры отсутствуют или неясны, пользователь должен согласовать этот вопрос с поставщиком. Как отмечено в разделе 7.8 приказа 8110.49, несоблюдение одобренных процедур может привести к серьезным последствиям, включая потенциальный отзыв сертификата типа воздушного судна [3].

Рекомендация пользователю 5: вносить изменения в модифицируемое пользователем ПО с использованием организованного и воспроизводимого процесса. Модификации такого про-

граммного обеспечения должны планироваться и определяться документированными требованиями. Процесс управления конфигурацией для модификации такого ПО должен включать управление изменениями, чтобы изменения были санкционированы, отслеживание изменений, чтобы они были задокументированы, идентификацию конфигурации, то есть обновление версии ПО или его обозначения, а также сопроводительной документации, выпуск и архивирование. Модификация должна быть верифицирована, предпочтительно с использованием тестирования на соответствие требованиям и контролироваться службой гарантии качества. Хотя такое ПО обычно не обязано соответствовать DO-178C, стоит рассмотреть применение процесса, похожего на DO-178C, например уровня D по DO-178C. Такой процесс помогает обеспечить функциональность, управление конфигурацией и гарантию качества такого ПО. Какой бы процесс ни использовался, он должен допускать аудит.

Рекомендация пользователю 6: при необходимости получать эксплуатационное одобрение модифицируемого пользователем ПО. Как уже отмечалось, некоторые виды модифицируемого пользователем ПО могут требовать одобрения со стороны органа эксплуатационного надзора, даже если им не требуется рассмотрение со стороны сертифицирующего органа по сертификации типа. Такое одобрение должно быть получено до использования измененного модифицируемого пользователем ПО в летной эксплуатации.

Рекомендация пользователю 7: обеспечивать правильную установку измененного модифицируемого пользователем ПО на воздушное судно. После получения соответствующего одобрения, например эксплуатационного, необходимо убедиться, что на воздушном судне фактически установлена именно одобренная версия модифицируемого пользователем ПО. Как отмечалось в рекомендации пользователю N 2, записи по техническому обслуживанию должны поддерживаться в актуальном состоянии.

Ссылки

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
2. Aeronautical Radio, Inc., *Guidance for the management of field loadable software*, ARINC Report 667-1 (Annapolis, MD: Airlines Electronic Engineering Committee, November 2010).
3. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Washington, DC: Federal Aviation Administration, Change 1, September 2011).
4. European Aviation Safety Agency, *Software aspects of certification*, Certification Memorandum CM-SWCEH-002 (Issue 1, August 2011).
5. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Washington, DC:

RTCA, Inc., December 2011).

*Квадратные скобки добавлены для пояснения.

†Следует отметить, что в некоторых источниках термин *airline certifiable software* также используется для обозначения программного обеспечения, которое тоже требует одобрения со стороны сертифицирующего органа, например STC, получаемого авиакомпанией. Поэтому этот термин следует использовать с осторожностью.

*Эта концепция также обсуждается в часто задаваемом вопросе номер 7 документа DO-248C [5].

Глава 20. Операционные системы реального времени

Сокращения

Сокращение	Расшифровка
AC	Консультативный циркуляр
AFDX	Коммутируемый полнодуплексный Ethernet авионики ARINC 664
APEX	Исполнительная среда приложения
API	Интерфейс прикладного программирования
BSP	Пакет поддержки платы
CAN	Сеть контроллеров
CEO	Главный исполнительный директор
COTS	Готовое коммерческое ПО
CPU	Центральный процессор
EAL	Уровни доверия к оценке
EPROM	Стираемое программируемое постоянное запоминающее устройство
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
IEEE	Институт инженеров электротехники и электроники
IMA	Интегрированная модульная авионика
I/O	Ввод-вывод
ISR	Процедура обслуживания прерывания
MMU	Блок управления памятью
MOS	Операционная система модуля
POS	Операционная система раздела
POSIX	Переносимый интерфейс операционной системы
RAM	Оперативная память
PR	Сообщение о проблеме
RSC	Повторно используемый компонент ПО
RTOS	ОСРВ (операционная система реального времени, real-time operating system)
SAP	Порт доступа сопровождения

Сокращение	Расшифровка
SVA	Анализ уязвимостей ПО (software vulnerability analysis)
UNIX	UNIX (Uniplexed Information and Computing System)
VMM	Монитор виртуальных машин

20.1 Введение

С наступлением нового тысячелетия операционная система реального времени (real-time operating system, RTOS; ОСПВ) стала обычным компонентом авиационных систем. В этой главе объясняется, что такое ОСПВ, зачем она используется, как она вписывается в типичную авионическую систему, какая функциональность от нее требуется, какие вопросы необходимо решать при ее использовании и какие трудности, связанные с ОСПВ, могут возникнуть в будущем.

20.2 Что такое ОСПВ?

Х. М. Дейтел пишет:

Операционные системы прежде всего являются менеджерами ресурсов; основным ресурс, которым они управляют, – это вычислительное оборудование в виде процессоров, устройств хранения, устройств ввода-вывода, коммуникационных устройств и данных. Операционные системы выполняют множество функций, таких как реализация пользовательского интерфейса, совместное использование аппаратуры несколькими пользователями, предоставление пользователям возможности обмениваться данными, предотвращение взаимных помех между пользователями, планирование ресурсов между пользователями, поддержка ввода-вывода, восстановление после ошибок, учет использования ресурсов, поддержка параллельных операций, организация данных для безопасного и быстрого доступа и обработка сетевых взаимодействий [1].

В общем случае операционная система – это программное обеспечение, управляющее аппаратными ресурсами компьютера и предоставляющее контролируемый доступ одному или нескольким приложениям, выполняющимся на этом компьютере. ОСПВ общего назначения выполняет те же операции, но дополнительно специально спроектирована для запуска приложений с очень точными временными требованиями. Критичные по безопасности ОСПВ являются подмножеством ОСПВ общего назначения и, как правило, обладают следующими характеристиками: детерминированность, то есть предсказуемость; быстрота отклика, причем в гарантированные сроки; управляемость со стороны разработчика и интегратора ПО; надежность; отказобезопасность [2]. Эти характеристики будут подробнее рассмотрены при обсуждении желательных свойств ОСПВ, критичной по безопасности.

В этой главе и в литературе по операционным системам реального времени используются

следующие термины:[*]

- *Приложение*. Программное обеспечение, состоящее из задач или процессов, выполняющих заданную функцию на воздушном судне. Приложение может содержать один или несколько разделов [3].
- *Интерфейс прикладного программирования (API)*. Формально определенный набор программных вызовов и процедур, к которому может обращаться прикладная программа для доступа к поддерживающим системным или сетевым службам.
- *Раздел*. «Программа, включая исполняемый код и данные, которая может быть загружена в единое адресное пространство в основном модуле» [3]. ОСРВ управляет использованием каждым разделом вычислительных ресурсов, то есть процессорного времени, памяти и других ресурсов, чтобы изолировать каждый раздел от всех остальных разделов, совместно использующих общее вычислительное оборудование.
- *Межраздельное взаимодействие*. Обмен между разделами.
- *Надежное обособление*. «Механизм, обеспечивающий требуемую изоляцию независимых эксплуатационных функций воздушного судна, размещенных в общих вычислительных ресурсах, при любых обстоятельствах, включая аппаратные ошибки и ошибки программирования. Цель надежного обособления – обеспечить тот же уровень функциональной изоляции, что и при федеративной реализации, то есть когда приложения размещаются по отдельности на независимых вычислительных элементах. Это означает, что надежное обособление должно поддерживать совместное сосуществование приложений на общем процессоре, одновременно гарантируя предотвращение несанкционированного или непреднамеренного взаимного влияния» [3].

20.3 Зачем использовать ОСРВ?

Операционная система реального времени, или ОСРВ, – это сердце многих современных авионических систем. ОСРВ может влиять на надежность программного обеспечения, производительность разработки и сопровождаемость [4]. Используя интерфейс прикладного программирования, ОСРВ предоставляет четко определенный и контролируемый интерфейс между приложением и лежащим ниже аппаратным обеспечением. Кроме того, ОСРВ сужает множество возможных взаимодействий, что упрощает верификацию корректности приложений. Без ОСРВ программисту необходимо детально знать лежащее ниже аппаратное обеспечение и особенности его функционирования.

В прошлом программное обеспечение встроенных систем писалось как монолитный массив кода и было тесно связано с лежащим ниже оборудованием. Это было необходимо, чтобы гарантировать выполнение требований по производительности. К сожалению, такой тип

проектирования делал ПО трудным в сопровождении, повторном использовании и переносе на другое оборудование.

Абстрагируя и инкапсулируя аппаратный интерфейс, ОСРВ в значительной степени устраняет необходимость для разработчика приложения писать аппаратно-зависимый код. ОСРВ повышает переносимость и экранирует от программиста сложность вычислительной системы, позволяя ему сосредоточиться на непосредственной задаче. Подробные знания о прерываниях, таймерах, аналого-цифровых преобразователях и тому подобном больше не требуются. В результате компьютер можно рассматривать как *виртуальную* машину, предоставляющую средства для безопасной, корректной, эффективной и своевременной работы. Иными словами, она облегчает жизнь, или по крайней мере делает ее легче [4].

Еще одним фактором роста использования ОСРВ стало повышение возможностей микропроцессоров. При сегодняшней скорости и емкости процессоров стало возможным запускать несколько приложений на одном процессоре. Кроме того, поскольку снижение массы уменьшает затраты на протяжении жизненного цикла воздушного судна, существует сильное стремление минимизировать количество оборудования на борту. Чтобы максимально использовать возможности процессора, уменьшить массу на воздушном судне и упростить техническое обслуживание, многие традиционные федеративные системы заменяются системами интегрированной модульной авионики (Integrated Modular Avionics, IMA). Такая архитектура позволяет нескольким приложениям выполняться на одном процессоре.

Для реализации интегрированной модульной авионики необходимо надежное обособление между приложениями. ОСРВ с надежным обособлением обеспечивает следующее [6]:

- Выполнение одного приложения не мешает выполнению любого другого приложения.
- Выделенные вычислительные ресурсы, назначенные приложениям, не конфликтуют и не приводят к столкновениям по памяти, расписанию или прерываниям.
- Общие вычислительные ресурсы распределяются между приложениями таким образом, чтобы сохранялись целостность ресурсов и обособление приложений.
- Ресурсы выделяются каждому приложению независимо от наличия или отсутствия других приложений.
- Предоставляются стандартизованные интерфейсы для приложений.
- Программные приложения и аппаратные ресурсы, необходимые для их размещения, независимы друг от друга.

Реализация интегрированной модульной авионики опирается на ОСРВ с надежным обособлением и ее окружение. Более подробно обособление рассматривается в главе 21.

Конкурентность авиационного рынка также сделала привлекательным применение готовых коммерческих ОСРВ (Commercial-Off-The-Shelf, COTS). Использование готовых коммерческих ОСРВ в значительной степени устраняет необходимость для системных разработчиков создавать собственную ОСРВ и поддерживать разработку операционной системы как ключевую компетенцию. Совместимая с DO-178C (КТ-178С) или DO-178В коммерческая операционная система стоит весьма дорого, но все же может оказаться экономически выгоднее, чем разрабатывать ОСРВ с нуля, доводить ее до уровня, пригодного для задач безопасности, разрабатывать ее в соответствии с DO-178С и затем сопровождать. Некоторые типичные проблемы готовых коммерческих ОСРВ рассматриваются далее в этой главе.

20.4 Ядро ОСРВ и поддерживающее его программное обеспечение

На рисунке 20.1 показано типичное место ОСРВ в авионической системе. Прикладное программное обеспечение может располагаться в одном или нескольких разделах. В системах, критичных по безопасности, применяются как ОСРВ с обособлением, так и без него. ОСРВ с обособлением становятся все более распространенными в авиационных проектах. Поскольку ядро ОСРВ и поддерживающее его ПО входят в состав встроенной системы, они должны удовлетворять DO-178C или DO-178B.[*] На рисунке 20.1 видно, что ядро ОСРВ поддерживается несколькими компонентами, включая библиотеки, пакет поддержки платы, драйверы устройств и интерфейс прикладного программирования. Ядро ОСРВ и каждый из основных поддерживающих компонентов кратко описаны в следующих подразделах.



20.4.1 Ядро ОСРВ

Ядро – это сердце ОСРВ. Оно предоставляет базовые службы, например службы стандарта ARINC 653, описанные в разделе 20.8.1, и проектируется по возможности независимо от лежащего ниже аппаратного обеспечения.[†] Цель большинства разработчиков ОСРВ состоит в том, чтобы изолировать ядро от аппаратуры для обеспечения повторного использования и переносимости. Аппаратно-зависимый код обычно реализуется в пакете поддержки платы и драйверах устройств.

20.4.2 Интерфейс прикладного программирования

Приложения получают доступ к службам ОСРВ через интерфейс прикладного программирования, который функционирует как интерфейс между ядром ОСРВ и приложением.

Интерфейс прикладного программирования предоставляет перечень доступных вызовов, которые программист может использовать при работе с ОСРВ. Исполнительная среда приложения (APEX) стандарта ARINC 653 является наиболее распространенным интерфейсом прикладного программирования, используемым в авионике для систем интегрированной модульной авионики. В некоторых ОСРВ также применяется переносимый интерфейс операционной системы (Portable Operating System Interface, POSIX). Стандарты POSIX, например IEEE 1003.1b, включают следующие определения интерфейсов для ОСРВ: управление задачами, асинхронный ввод-вывод, семафоры, очереди сообщений, управление памятью, очереди сигналов, планирование, часы и таймеры [7]. Этот интерфейс основан на UNIX, семействе операционных систем UNIX. Поскольку существовало несколько разновидностей UNIX, имеется и несколько стандартов IEEE, связанных с POSIX [4]. Так как UNIX не разрабатывался для жесткой среды реального времени, в авионике переносимый интерфейс операционной системы необходимо применять осторожно. Это было одной из причин разработки исполнительной среды приложения стандарта ARINC 653. Между стандартом ARINC 653 и POSIX есть определенное сходство, поскольку многие концепции этого интерфейса повлияли на ARINC 653. Различие между переносимым интерфейсом операционной системы и исполнительной средой приложения стандарта ARINC 653 невелико на уровне интерфейса прикладного программирования, однако на уровне организации программы различия уже гораздо существеннее. Исполнительная среда приложения задумывалась для обеспечения обособления между различными приложениями, использующими один и тот же процессор, то есть для обеспечения надежного обособления между разделами. Переносимый интерфейс операционной системы не предоставляет такого же типа надежного обособления, однако поддерживает совместную работу нескольких процессов. В таблице 20.1 приведено общее сравнение возможностей стандарта ARINC 653 и POSIX

на высоком уровне.

Таблица 20.1 Базовое сравнение возможностей стандарта POSIX и ARINC 653

POSIX	ARINC 653 APEX
Событийно-ориентированная модель выполнения.	Циклическая модель выполнения для планирования разделов.
Использует модель выполнения с несколькими процессами и несколькими потоками.	По сути используется та же идея, но терминология отличается: вместо процессов используются разделы, а вместо потоков – процессы.
Не поддерживает временное обособление.	Поддерживает временное обособление с циклическим планированием разделов по схеме кругового обслуживания (round-robin).
Поддерживает приоритетное вытесняющее планирование для процессов и потоков. Циклическое планирование также может быть реализовано программно.	Поддерживает приоритетное вытесняющее планирование для процессов внутри раздела и циклическое планирование между разделами.
Планирование ввода-вывода зависит от драйверов устройств и может быть либо по прерываниям, либо опросным.	Возможны разные модели, в том числе управление по прерываниям, пока раздел, выполняющий ввод-вывод, активен, либо опросный ввод-вывод для более жесткого управления запросами.
Поддерживает изоляцию памяти.	По сути реализует ту же цель, но использует пространственное обособление вместо термина изоляция памяти. Разметка памяти, политики доступа, кэш-политики и механизмы защиты памяти задаются статическими конфигурационными таблицами.
Использует сокеты.	Использует выборочные и очередные порты, связанные с псевдопортами для обмена с внешними системами. Они могут быть связаны с сокетами Ethernet или другими протоколами.

POSIX	ARINC 653 APEX
Использует межпроцессное взаимодействие.	Обменивается данными через выборочные и очередные порты либо через общую память, доступ к которой контролируется системой управления памятью.
Использует таймеры.	Сервисы Интерфейс прикладного программирования может реагировать на тайм-ауты. Процессы могут быть периодическими; механизмы задержки и другие механизмы помогают синхронизировать прикладной код по времени.
Использует обработку сигналов и требует, чтобы пользователь сам задавал политики реакции на ошибки.	Использует контроль состояния для управления пользовательскими политиками, заданными в конфигурационных таблицах, либо передает обработку в определенные пользователем обработчики ошибок.

20.4.3 Пакет поддержки платы

Пакет поддержки платы (board support package, BSP) изолирует ОСРВ от вычислительной аппаратуры целевого вычислителя. Он позволяет ядру ОСРВ быть переносимым на различные аппаратные архитектуры в пределах одного семейства центральных процессоров. «Пакет поддержки платы инициализирует процессор, устройства и память; выполняет различные проверки памяти и так далее. После завершения инициализации пакет поддержки платы может продолжать выполнять низкоуровневые манипуляции с кэшем» [9], а также некоторый другой доступ к аппаратуре, например операции с флэш-памятью или доступ к таймерам. Значительная часть кода пакета поддержки платы работает в привилегированном режиме и тесно взаимодействует с ОСРВ. Пакет поддержки платы иногда называют *слоем аппаратных абстракций*, *системой аппаратного интерфейса* или *ПО поддержки платформы*. Он настраивается под конкретное оборудование и часто реализуется на языке С и на ассемблере. Иногда пакет поддержки платы разрабатывается пользователями ОСРВ, например разработчиком авионики, на основе шаблона от поставщика ОСРВ, особенно когда в авионике используется нестандартное оборудование. В других случаях пакет поддержки платы предоставляется и/или адаптируется поставщиком ОСРВ. В зависимости от характера оборудования пакет поддержки платы может быть довольно крупным компонентом.

20.4.4 Драйвер устройства

Подобно пакету поддержки платы, драйвер устройства предоставляет интерфейс между ядром ОСРВ и аппаратным устройством. Аппаратные устройства могут находиться либо на процессорной плате, либо отдельно от нее. Оборудование на процессорной плате может обслуживаться пакетом поддержки платы или отдельным драйвером. Драйвер представляет собой низкоуровневый, аппаратно-зависимый программный модуль, обеспечивающий интерфейс между приложением, иногда через библиотечные или ядерные функции ОСРВ более высокого уровня, и конкретным аппаратным устройством. Драйвер устройства отвечает за доступ к аппаратным регистрам устройства и может включать обработчик прерываний для обслуживания прерываний, генерируемых устройством. Большинство авионических систем содержит несколько устройств, например устройства Ethernet, оконечные системы коммутируемого полнодуплексного Ethernet авионики ARINC 664 (AFDX), последовательные устройства RS232, порты ввода-вывода, аналого-цифровые преобразователи и шины данных CAN.

20.4.5 Поддерживающие библиотеки

Нередко языково-специфическая поддержка времени выполнения предоставляется вместе с ядром ОСРВ в виде библиотек. Например, «в языке С ряд спецификаций стандартных библиотек позволяет пользователю вызывать функции для перемещения памяти, сравнения строк и использования математических функций, таких как *mod*, *floor* и *cos* » [9]. Эти библиотеки времени выполнения часто поставляются вместе с ОСРВ, либо внутри ядра, либо как отдельный библиотечный пакет. Обычно библиотеки предоставляются поставщиком ОСРВ в виде предварительно скомпилированного объектного кода, который может компоноваться вместе с приложениями. Благодаря этому подключаются только те библиотечные функции, которые действительно нужны. Некоторые компоновщики вообще способны определить, какие библиотечные функции вызываются приложением, и включить в исполняемый файл только их.

20.5 Характеристики ОСРВ, используемой в системах, критичных по безопасности

В этом разделе перечислены ключевые характеристики ОСРВ, критичных по безопасности и применяемых в авиационной области.

20.5.1 Детерминированность

Система, критичная по безопасности, выдает правильный ответ, в правильном порядке и в правильное время. «Детерминированность – это характеристика системы, позволяющая корректно предсказать ее будущее поведение, исходя из текущего состояния и знания будущих изменений ее окружения, то есть входных воздействий» [10]. Недетерминированность

же, напротив, «означает, что будущее поведение системы нельзя корректно предсказать. Непредсказуемую систему нельзя назвать безопасной» [10]. Поведение ОСРВ в критичной по безопасности области должно быть предсказуемым. Иными словами, при заданном входе ОСРВ выдает один и тот же выход. В частности, выходные результаты должны оставаться в пределах, определенных требованиями. Детерминированная ОСРВ обеспечивает как функциональную корректность, так и временную корректность, определенные требованиями, а также потребляет только известные и ожидаемые, то есть ограниченные, объемы памяти и времени.

20.5.2 Надежная производительность

ОСРВ должна удовлетворять требованиям к производительности, необходимой для того, чтобы приложение могло выполнять свою предназначенную функциональность. Производительность ОСРВ определяется многими факторами, включая времена вычислений, методы планирования, обработку прерываний, времена переключения контекста, управление кэшем, диспетчеризацию задач и так далее [4]. Показатели производительности обычно приводятся в техническом описании ОСРВ. Производительность сильно меняется в зависимости от конкретного оборудования, интерфейсов, тактовой частоты, компилятора, проекта и рабочей среды. Поэтому при выборе ОСРВ компании почти всегда проводят собственные тесты производительности, прежде чем решить, какую ОСРВ покупать.

20.5.3 Совместимость с аппаратным обеспечением

Ядро ОСРВ должно быть совместимо с выбранным процессором. Аналогично, пакет поддержки платы должен быть совместим с выбранной процессорной платой и основными устройствами, а драйверы устройств – с устройствами системы.

20.5.4 Совместимость со средой

ОСРВ должна поддерживать выбранный язык программирования и компилятор. Большинство ОСРВ поставляется с интегрированной средой разработки, включающей компилятор, компоновщик, отладчик и другие инструменты, необходимые для успешной интеграции приложений, ОСРВ, поддерживающего программного обеспечения и аппаратуры.

20.5.5 Устойчивость к неисправностям

Устойчивость к неисправностям – это «встроенная способность системы продолжать выполнение при наличии ограниченного числа неисправностей аппаратуры или ПО» [11]. ОСРВ, устойчивая к неисправностям, предусматривает возможность сбоев приложений и предоставляет механизмы восстановления или останова. ОСРВ обеспечивает управление неисправностями, которое включает: (1) обнаружение неисправностей, отказов и ошибок; (2) корректную идентификацию неисправностей, отказов и ошибок при их обнаружении; и (3) выполнение заранее предусмотренной для системы реакции [11].

20.5.6 Контроль состояния

Контроль состояния тесно связан с управлением неисправностями и является еще одной особенностью ОСПВ, устойчивой к неисправностям. Контроль состояния – это служба, часто предоставляемая ОСПВ для управления неисправностями большей части, если не всей системы, использующей ОСПВ. В DO-297 объясняется, что контроль состояния отвечает за обнаружение, изоляцию, сдерживание и регистрацию отказов, которые могут неблагоприятно воздействовать на ресурсы или приложения, использующие эти ресурсы [11]. DO-297 сосредоточен на платформе интегрированной модульной авионики, однако ОСПВ играет ключевую роль в общей схеме управления состоянием системы. В исследовательском отчете FAA по ОСПВ в системах интегрированной модульной авионики сказано следующее:

Контроль состояния определен в ARINC 653. Монитор состояния – это функция ОСПВ, отвечающая за выявление, реакцию на аппаратные отказы, отказы приложений и отказы ОСПВ, а также за их регистрацию. Контроль состояния помогает изолировать неисправности и не допускать распространения отказов.

За счет классификации неисправностей и реакций управления состоянием появляется широкий набор вариантов, который помогает поставщику приложения и интегратору системы интегрированной модульной авионики выбирать подходящее поведение. Для описания предполагаемого восстановления при выявленных неисправностях используются таблицы конфигурации, например игнорирование неисправности, повторная инициализация процесса, теплый или холодный перезапуск раздела, выполнение полного сброса системы либо вызов заданной системой процедуры для выполнения заданных системой действий [12].

ARINC 653 определяет ряд кодов ошибок, которые должны обнаруживаться и обрабатываться. «Ошибка может обрабатываться внутри процесса, в разделе либо в модуле или процессе контроля состояния» [9].

Цель функции контроля состояния в ARINC 653 состоит в том, чтобы локализовать неисправности до того, как они распространятся через границу интерфейса. Помимо методов самоконтроля, в ОСПВ сообщаются нарушения со стороны приложений, сбои связи и неисправности, обнаруженные приложениями. Для задания действий, которые следует выполнить в ответ на конкретную неисправность, используется таблица восстановления по неисправностям. Это действие инициируется монитором состояния и может включать завершение приложения и запуск альтернативного приложения вместе с соответствующим уровнем отчетности [13].[*]

Действие по восстановлению зависит от проекта системы и требований.

20.5.7 Пригодность к сертификации

Пригодность к сертификации является необходимой характеристикой ОСПВ, используемой в системах, критичных по безопасности. Если ОСПВ предполагается устанавливать на воздушное судно, ей потребуется удовлетворять целям DO-178C.[†] Стоит отметить, что сама ОСПВ не сертифицируется; вместо этого она разрабатывается так, чтобы быть сертифицируемой как часть воздушного судна, двигателя или воздушного винта.[‡] Для большинства коммерческих ОСПВ, пригодных для установки в авиационные изделия, доступен *сертификационный пакет*, поддерживающий общую сертификацию. Сертификационный пакет содержит артефакты, демонстрирующие соответствие DO-178C или DO-178B и поддерживающие безопасность и сертификацию. Некоторые типичные проблемы сертификации ОСПВ рассматриваются далее в этой главе, в разделе 20.7.

20.5.8 Сопровождаемость

Сопровождаемость – желательная характеристика для любого программного обеспечения, используемого в системах, критичных по безопасности, включая ОСПВ. Поскольку жизненный цикл системы, вероятно, будет очень долгим, возможно более 20 лет, ОСПВ должна быть сопровождаемой, то есть допускающей модификации для учета дополнительных приложений, оборудования, аппаратуры и тому подобного. Данные ЖЦ ПО и процессы управления конфигурацией, требуемые DO-178C, поддерживают сопровождаемость.

20.5.9 Повторное использование

Хотя это и не является обязательной характеристикой, большинство ОСПВ проектируется с расчетом на повторное использование. Как отмечалось ранее, ядро абстрагируется от аппаратного обеспечения посредством пакета поддержки платы и драйверов устройств. Консультативный циркуляр FAA AC 20-148 под названием *Reusable Software Components* обычно применяется и к ОСПВ. AC 20-148 содержит рекомендации FAA по упаковке повторно используемого компонента ПО (RSC) для применения в нескольких системах. Даже если проект не стремится получить от FAA письмо по RSC, он, вероятно, все равно захочет следовать рекомендациям этого циркуляра, чтобы сделать компонент ОСПВ максимально пригодным для повторного использования. Более подробно повторное использование рассматривается в главе 24.

20.6 Функции ОСПВ, используемой в системах, критичных по безопасности

В этом разделе перечислены ключевые функции большинства ОСПВ, критичных по безопасности и применяемых в авиационной области. Эти функции тесно связаны с отмеченными ранее характеристиками и обеспечивают средства для реализации этих характеристик. Рассматриваются функции, наиболее существенные с точки зрения безопасности.

20.6.1 Многозадачность

Многозадачность – это метод, при котором несколько задач, также называемых процессами, совместно используют общий процессор. Многозадачность создает видимость того, что на процессоре одновременно выполняется много задач, хотя на самом деле ядро чередует их выполнение (для однопроцессорной системы) с помощью алгоритма планирования. Когда процессор прекращает выполнение одной задачи и начинает выполнять другую, это называется переключением контекста. Если переключения контекста происходят достаточно часто, создается впечатление, что задачи выполняются параллельно. Каждая кажущаяся независимой программа (задача) имеет собственный контекст, то есть процессорное окружение и системные ресурсы, которые задача «видит» каждый раз, когда ядро назначает ее на выполнение.

20.6.2 Гарантированная и детерминированная планируемость

Соблюдение предельных сроков является одним из самых фундаментальных требований к ОСПВ. Для систем, критичных по безопасности, планирование и своевременное завершение нескольких задач должны быть детерминированными. Для ОСПВ, применяемых в современной авионике, например в системах интегрированной модульной авионики, обычно требуется два вида планируемости: [*] (1) планирование между разделами и (2) планирование внутри разделов. Ниже рассматриваются оба случая.

20.6.2.1 Планирование между разделами

Модель ARINC 653 требует циклической последовательности выполнения между разделами по схеме round-robin (циклическое обслуживание). За счет этого формируется временной интервал, называемый большим временным кадром. Внутри большого временного кадра определяются временные слоты фиксированной длительности. При этом длительность каждого слота не обязана быть одинаковой. Таблица конфигурации определяет, какие разделы будут выполняться в каком временном слоте внутри кадра:

Раздел может быть назначен более чем одному слоту в пределах кадра. Выполнение каждого раздела идет последовательно, начиная с начала кадра. Когда выполнен последний раздел в кадре, последовательность повторяется. Временные слоты жестко обеспечиваются MOS (операционной системой модуля). Для своевременного переключения между разделами используется тактовое устройство [9].[†]

20.6.2.2 Планирование внутри разделов

Внутри раздела может существовать множество схем планирования. Большинство ОСПВ осуществляет планирование на основе приоритетов и вытеснения. Каждой задаче назначается приоритет. Задача с более высоким приоритетом выполняется раньше всех задач с более низким приоритетом. Под *вытесняющим* понимается то, что как только задача с

более высоким приоритетом готова к выполнению, она может остановить выполнение задачи с более низким приоритетом, которая исполняется в данный момент. Это похоже на ситуацию, когда вы находитесь на совещании с начальником. Если появляется генеральный директор компании, ваше обсуждение с начальником прерывается, пока он или она не закончит более приоритетное обсуждение с генеральным директором. Следующий список кратко описывает наиболее распространенные подходы к планированию, применяемые в ОСРВ для систем, критичных по безопасности. По каждому из этих процессов существует большой объем литературы; здесь приводится лишь краткий обзор:[‡]

- *Циклический диспетчер*. Эта техника циклически проходит по набору процессов, порядок выполнения которых заранее определен. Это распространенный подход даже в тех случаях, когда ОСРВ вообще не используется.
- *Round-robin (циклическое обслуживание)*. Название алгоритма происходит от принципа round-robin, при котором каждый получает равную долю чего-либо. Автор однажды участвовал в круговом конкурсе выставочного показа в 4-Н. Каждому участнику давалось одинаковое время, чтобы показать лошадь, бычка, свинью и ягненка. Как сообщается, автор получил приз Reserved Grand Champion. Алгоритм планирования round-robin является одним из самых простых алгоритмов распределения процессорного времени. Система определяет малый временной квант, а все процессы хранятся в кольцевой очереди. Планировщик проходит по очереди, выделяя каждому процессу ресурсы процессора на интервал одного кванта времени. По мере поступления новых процессов они добавляются в хвост очереди. Планировщик работает так: выбирает первый процесс из очереди, устанавливает таймер, который прерывает выполнение после одного кванта времени, а затем передает управление процессу. Если к концу кванта процесс не завершен, он вытесняется и помещается в хвост очереди. Если же процесс завершается до конца кванта, он освобождает процессор. Переключение контекста происходит каждый раз, когда процессу предоставляется доступ к процессору. Переключение контекста добавляет накладные расходы ко времени выполнения процесса [9].
- *Вытесняющее планирование с фиксированными приоритетами*. Каждая задача имеет фиксированный приоритет, который не меняется. При таком подходе планировщик гарантирует, что в любой момент времени из всех задач, готовых к выполнению, исполняется задача с наивысшим приоритетом. Этот подход использует прерывание для вытеснения задачи с более низким приоритетом, если задача с более высоким приоритетом становится готовой. Преимущество такого подхода в том, что он не позволяет задачам низкого приоритета монополизировать процессорное время. Однако отрицательная сторона состоит в том, что задача низкого приоритета может быть заблокирована от выполнения на неопределенно долгое время. Многие ОСРВ поддерживают эту

схему планирования.

- *Планирование по монотонной частоте.* Алгоритм rate monotonic scheduling (планирование по монотонной частоте) считается оптимальным алгоритмом статических приоритетов. Это вытесняющий алгоритм с приоритетами, который назначает задачам фиксированные приоритеты таким образом, чтобы максимизировать планируемость и гарантировать соблюдение всех предельных сроков. Каждой задаче назначается приоритет в соответствии с ее периодом. Чем короче период, то есть чем выше частота, тем выше приоритет. Этот алгоритм планирования реализован в ряде коммерческих ОСРВ для систем, критичных по безопасности.
- *Планирование по монотонному сроку.* Алгоритм deadline monotonic scheduling (планирование по монотонному сроку) похож на планирование по монотонной частоте, за исключением того, что приоритет назначается задаче с наименьшим предельным сроком, а не с наименьшим периодом. Соответственно, процесс с самым близким предельным сроком получает наивысший приоритет, а процесс с самым далеким – наименьший.
- *Планирование по ближайшему сроку завершения.* Это вытесняющая политика с динамическими приоритетами. Она помещает задачи в очередь приоритетов, так что всякий раз, когда задача завершается, в очереди ищется процесс, ближайший к своему сроку завершения.[*] Этот процесс и становится следующей задачей, назначаемой на выполнение. По сути, планировщик выбирает для выполнения процесс с наиболее ранним сроком завершения, который вытесняет любые процессы с более поздними сроками.
- *Планирование по наименьшему запасу времени (также известное как least laxity first, также называемое планированием по наименьшей слабине).* Как и подход сначала ближайший срок (earliest deadline first), это вытесняющая политика с динамическими приоритетами. Приоритет назначается на основе запаса времени (оставшееся время до предельного срока минус оставшееся время выполнения, которое бывает трудно точно предсказать). Запас времени можно также описать как разность между временем до срока завершения задачи и ее оставшейся потребностью во времени обработки. Процесс с наименьшим запасом времени вытесняет процессы с большим запасом.

20.6.3 Детерминированное взаимодействие между задачами

Поскольку в каждый момент времени может выполняться только одна задача (для одноядерного процессора), должны существовать механизмы, позволяющие задачам обмениваться информацией друг с другом. Во многих ОСРВ очереди и сообщения предоставляют средство межзадачного взаимодействия; такой подход помогает избежать ситуации, когда одна задача читает участок памяти в тот момент, когда другая в него записывает. Существует как минимум три причины, по которым требуется межзадачное взаимодействие [4]:

1. *Для синхронизации или координации действий без передачи данных.* Обычно такие задачи связаны событиями, включая факторы, связанные со временем, например задержки или истекшее время. Для синхронизации общих ресурсов используется синхронизация задач или кооперация задач. Между синхронизацией задач и координацией задач имеется некоторое пересечение, поскольку синхронизированные операции задач могут использоваться для координации задач. Однако как только для блокировки и освобождения задач используется прерывание, речь уже идет не о координации задач, потому что задачи синхронизируются с прерываниями реального времени.
2. *Для обмена данными без синхронизации.* Бывают случаи, когда задачи обмениваются информацией без необходимости синхронизации или координации. Это часто реализуется с помощью хранилища данных, например пулов или буферов, в которое встроено взаимное исключение, гарантирующее, что данные не будут повреждены.
3. *Для обмена данными в строго синхронизированные моменты времени.* Это сценарий, при котором задачи ожидают событий и используют данные, связанные с этими событиями. Например, синхронизация задач может достигаться путем приостановки или остановки задач до тех пор, пока не будут выполнены необходимые условия.

20.6.4 Надежное управление памятью

ОСРВ поддерживают выделение памяти и отображение памяти и предпринимают действия, когда задача использует память. Большинство процессоров включает встроенный блок управления памятью (memory management unit, MMU), который позволяет программным потокам работать в аппаратно защищенном адресном пространстве [14]. Блок управления памятью отвечает за обработку обращений к памяти. Его функциональность обычно включает преобразование виртуальных адресов в физические, защиту памяти, управление кэшем и арбитраж шины. Поскольку блок управления памятью является функциональностью готового коммерческого компонента, поставляемой вместе с процессором, а его целостность заранее неизвестна, в ходе сертификации ему обычно требуется уделять особое внимание. Как правило, корректность работы блока управления памятью, как и общая функциональность процессора, проверяется посредством детального тестирования ОСРВ и поддерживающего ее программного обеспечения, например пакета поддержки платы. Некоторые типичные проблемы памяти, которых следует избегать, такие как утечки памяти, фрагментация памяти и когерентность памяти, обсуждаются далее.

20.6.5 Обработка прерываний

Системы реального времени обычно реагируют на внешние события с помощью прерываний через процедуры обслуживания прерываний (interrupt service routine, ISR). Обычно ОСРВ сохраняет контекст прерванной задачи, чтобы после обработки прерывания вернуть-

ся к ней. Обработка прерываний может существенно влиять на производительность системы. В общем случае обработка прерываний делает следующее [15]:

- Приостанавливает активную задачу.
- Сохраняет относящиеся к задаче данные, которые потребуются при возобновлении задачи. Передает управление соответствующей процедуре обслуживания прерывания.
- Выполняет некоторую обработку в процедуре обслуживания прерывания для определения необходимых действий. Извлекает и сохраняет критические входные данные, связанные с прерыванием. Устанавливает требуемые специфичные для устройства выходные значения.
- Сбрасывает аппаратуру прерывания, чтобы могло быть распознано следующее прерывание. Передает управление следующей задаче, определенной планировщиком.

Чтобы предотвратить состояния гонки, то есть ситуацию, когда две задачи без координации пытаются изменить одни и те же данные, ОСРВ иногда отключают прерывания во время доступа к внутренним критическим структурам данных операционной системы или их модификации. Максимальное время, в течение которого ОСРВ отключает прерывания, называется латентностью прерывания. Худшее время латентности прерывания должно включать «все программные накладные расходы, которые необходимо выдержать до фактического выполнения процедуры обслуживания прерывания» [16]. Обычно существует компромисс между латентностью прерывания, пропускной способностью и использованием процессора.[*] Это следует учитывать при определении наихудшей производительности [17]. Ключом к обеспечению производительности является низкая латентность прерывания [18].

Некоторые ОСРВ уменьшают латентность прерываний с помощью техники, называемой *отложенной обработкой работы*. Если процедура обслуживания прерывания приводит к необходимости запланировать другую работу ОСРВ во время обработки прерывания, эта работа может быть сохранена в очереди работ и вызвана позже. Это сокращает длительность критических участков и делает систему более отзывчивой; однако добавляет сложность при расчете наихудшего времени выполнения.

20.6.6 Функции-перехватчики

Многие ОСРВ включают так называемые функции-перехватчики (*hook functions*) или функции обратного вызова (*callback functions*). Функция-перехватчик позволяет разработчику связать прикладной код с определенной функцией внутри ОСРВ. Перехватчик выполняется функцией ОСРВ, иногда с повышенным приоритетом и доступом к ресурсам ОСРВ. Эти перехватчики могут использоваться для расширения ОСРВ под конкретные потребности пользователя, особенно если исходный код проприетарной ОСРВ недоступен. Функции-

перехватчики позволяют настраивать поведение без изменения исходного кода ОСРВ и без риска ненамеренно внести дефект. Они могут позволять пользователю ОСРВ выполнять некоторые действия до или после того, как ОСРВ отреагирует на некоторое событие, без накладных расходов на создание отдельной задачи. Нередко функции-перехватчики используются для помощи при отладке в ходе разработки. В конечном продукте они могут быть деактивированы или отключены. Очевидно, в системах, критичных по безопасности, с функциями-перехватчиками нужно обращаться очень осторожно, поскольку они способны изменять поведение самой ОСРВ [19].

20.6.7 Проверка робастности

ОСРВ должна проектироваться так, чтобы защищать себя от определенных ошибок пользователя. Примеры встроенных проверок робастности включают проверку параметров, передаваемых через интерфейс прикладного программирования приложением, выполняющим системный вызов, проверку того, что приоритет задачи находится в допустимом диапазоне ОСРВ, или проверку того, что операция над семафором выполняется только над объектом-семафором. Проверка робастности сложна, и верификация функций робастности может требовать специальных методов тестирования [20].

20.6.8 Файловая система

Файловая система – это способ управления хранением данных.[†] Подобно файловой системе в настольной среде, файловая система ОСРВ управляет различными формами хранения данных на аппаратуре и скрывает их детали. Файловая система предоставляет возможность открывать, закрывать, читать, записывать и удалять файлы и каталоги.

Реализация зависит от типа носителя данных, например оперативной памяти RAM, флеш-памяти, стираемого программируемого постоянного запоминающего устройства EPROM или сетевых носителей. Файловая система позволяет нескольким разделам получать доступ к носителю данных. Ядро ОСРВ или поддерживающая его библиотека реализует файловую систему, чтобы управлять низкоуровневыми деталями носителя для разделов, которые им пользуются. Авионические приложения часто используют файловую систему для хранения и извлечения данных. Например, системы управления полетом и системы предупреждения о рельефе местности могут использовать файловую систему для доступа к своим базам данных [13].

20.6.9 Надежное обособление

Многие ОСРВ для интегрированной модульной авионики поддерживают надежное обособление. Определение *надежного обособления* уже приводилось ранее, в разделе 20.2. Раздел 2.3.3 DO-297 поясняет следующее:

Цель надежного обособления состоит в обеспечении эквивалентного уровня функциональ-

ной изоляции и независимости, как и при реализации федеративной системы, то есть когда приложения по отдельности размещаются на отдельных LRU (line replaceable unit, линейно заменяемый блок). Это означает, что надежное обособление поддерживает совместное сосуществование приложений, использующих общие ресурсы, одновременно гарантируя, что любая попытка несанкционированного или непреднамеренного взаимодействия будет обнаружена и ослаблена. Механизмы защиты надежного обособления на платформе независимы от любых размещенных приложений, то есть приложения не могут изменить защиту обособления, обеспечиваемую платформой [11].

ОСРВ играет ключевую роль в реализации надежного обособления, чтобы обеспечить защиту общих ресурсов времени, памяти и ввода-вывода. Более подробно обособление описывается в главе 21.

20.7 Вопросы, которые следует учитывать при работе с ОСРВ

В этом разделе кратко рассматриваются технические вопросы и вопросы сертификации, которые часто требуют внимания при разработке ОСРВ и при ее внедрении в систему, критичную по безопасности. Некоторые из этих вопросов уже упоминались ранее, но здесь они раскрываются подробнее. Это не исчерпывающий перечень проблем, которые могут возникнуть в проекте, а лишь часть тех, что традиционно встречались на практике и обычно наиболее существенны с точки зрения безопасности.

20.7.1 Технические вопросы, которые следует учитывать

20.7.1.1 Конфликты за ресурсы

Как следует из самого названия, конфликт за ресурс – это конфликт из-за общего ресурса, например процессора или памяти. В ОСРВ необходимо учитывать три конкретных вида таких конфликтов: взаимную блокировку, голодание и недопуск. Каждый из них описан ниже:

- *Взаимная блокировка* – это состояние, при котором ни один процесс не сможет завершиться, потому что процессы не могут получить доступ к ресурсам, необходимым им для продвижения вперед. Чтобы возникла взаимная блокировка, должны выполняться следующие условия: (1) взаимное исключение (ресурсы могут выделяться только одному процессу одновременно), (2) удержание и ожидание (процесс может захватить один ресурс и ждать другие), (3) отсутствие вытеснения (ресурс нельзя принудительно отобрать) и (4) циклическое ожидание (процессы удерживают ресурсы, которые нужны другим процессам). Взаимную блокировку почти невозможно обнаружить тестированием, но ее можно выявить анализом (формальные методы могут в этом помочь). Обычно взаимной блокировки избегают на уровне проектирования [7], предотвращая одно из четырех указанных условий в архитектуре ОСРВ [21]. *Голодание* возникает «когда

задача не получает достаточных ресурсов, чтобы завершить обработку в отведенное ей время», потому что нужные ресурсы используются другими задачами [22].

- *Недопуск* – это особый случай голодания, когда задача оказывается *не допущена* к выполнению, поскольку другая задача была заблокирована до возврата общего ресурса [7].

20.7.1.2 Инверсия приоритетов

Инверсия приоритетов – это вид взаимной блокировки, возникающий, когда задача высокого приоритета вынуждена ждать освобождения общего ресурса, принадлежащего задаче более низкого приоритета. «Период времени, в течение которого задача удерживает блокировку общего ресурса, называется критической секцией или критической областью этой задачи» [21]. Известный пример инверсии приоритетов – миссия Mars Pathfinder. Через несколько дней после начала миссии Pathfinder начал регулярно перезагружаться, из-за чего система надолго становилась недоступной. Тестирование и анализ показали, что проблема была вызвана инверсией приоритетов. Задача программного обеспечения низкого приоритета на Pathfinder делила ресурс с задачей высокого приоритета. Задача низкого приоритета заблокировала общий ресурс после того, как была вытеснена несколькими задачами среднего приоритета. «Когда другая задача высокого приоритета обнаружила, что предыдущая задача высокого приоритета не завершилась, она инициировала перезагрузку системы» [5]. Инверсия приоритетов стала возможной из-за глобальной настройки ОСРВ по умолчанию.

Обычно для решения проблемы инверсии приоритетов применяют два подхода: (1) протокол наследования приоритета или (2) протокол потолка приоритетов. Некоторые ОСРВ предоставляют оба протокола и позволяют пользователю выбрать предпочтительный алгоритм [14]. Kyle и Bill Renwick рассматривают достоинства и недостатки обоих протоколов в своей статье *How to use priority inheritance*. Они утверждают, что «лучшая стратегия решения проблемы инверсии приоритетов состоит в том, чтобы спроектировать систему так, чтобы инверсия вообще не могла возникнуть» [23]. По теме инверсии приоритетов существует обширная литература. В этом разделе дано лишь введение.

20.7.1.3 Утечки памяти

Утечка памяти – это «ошибка в логике выделения динамической памяти программой, из-за которой освобождаемая память не возвращается, что в итоге приводит к отказу вследствие исчерпания памяти» [24]. Обычно с этим борются, избегая динамического управления памятью в приложениях, критичных по безопасности; для этого после инициализации механизм выделения памяти блокируется, а освобождение памяти запрещается.

20.7.1.4 Фрагментация памяти

Фрагментация памяти возникает, когда память используется неэффективно, что приводит к снижению производительности и, возможно, к исчерпанию памяти. Чтобы этого избежать, техника распределения памяти должна быть четко определена, организована и контролируема. Следующие практики помогают избежать фрагментации памяти: (1) фиксировать размер выделяемых блоков, (2) разделять и задавать размеры выделяемой памяти, (3) использовать идентификаторы для отслеживания выделенной памяти и (4) защищать и изолировать сегменты [4].

20.7.1.5 Межзадачные помехи

Межзадачные помехи возникают, когда одна задача может изменять память другой задачи или даже самой операционной системы. Это устраняется путем отделения ОСРВ от приложений (например, с использованием защищенных режимов) и за счет применения возможностей блока управления памятью [4]. Как отмечалось ранее, защита памяти критически важна при реализации надежного обособления.

20.7.1.6 Джиттер

Кэш и конвейерные возможности процессора повышают производительность, но одновременно добавляют неопределенность во время выполнения задач. «Эта неопределенность называется джиттером, и аналитически ее практически невозможно количественно оценить» [25]. Джиттер зависит от аппаратной платформы, подхода к планированию в операционной системе и задач, совместно использующих процессор. Распространенным способом борьбы с джиттером кэша является выборочная очистка кэша. Кэш очищается при переключении разделов, чтобы входящий раздел в начале своего интервала имел чистую кэш-память. «Очистка означает копирование всех значений, имеющихся только в кэше, обратно в основную память (то есть они были обновлены и используется режим обратной записи). Это переносит накладные расходы на начало работы раздела, вместо того чтобы распределять их по всему интервалу» [9]. Время, требуемое для выполнения очистки, меняется в зависимости от количества значений, которые необходимо записать в память [9].

20.7.1.7 Уязвимости

Поскольку ОСРВ обычно разрабатывается отдельно от приложений, которые ее используют, она может содержать уязвимости, о которых пользователи должны знать. Для выявления и ослабления уязвимостей ОСРВ необходим Анализ уязвимостей ПО (SVA). Любые уязвимости, которые не устраняются проектированием ОСРВ, должны быть выявлены для системного интегратора и (или) разработчиков приложений, чтобы те могли их учесть. Кроме того, необходимо определить все предположения и ограничения, относящиеся к пользователям ОСРВ, чтобы гарантировать ее корректное применение. Некоторые опасные факторы

могут потребовать специальных практик проектирования или кодирования на стороне приложений. «Другие опасные факторы могут ослабляться специальным анализом или специальной техникой верификации, выполняемой на последующих этапах процесса верификации» [20].

Анализ уязвимостей ПО (SVA, software vulnerability analysis) может выявить области потенциальных аномалий, которые могут использоваться как вход не только для плана проверки робастности или стресс-тестирования, но и для анализа функциональных опасностей системы или для оценки безопасности системы (system safety assessment, SSA). То, как именно проводится SVA, определяется разработчиком ОСПВ или заявителем [26].[*]

Анализ уязвимостей ПО (SVA) основывается на конкретной реализации ОСПВ. Исследовательский отчет FAA *Study of Commercial Off-The-Shelf (COTS) Real-Time Operating Systems (OSPB) in Aviation Applications* содержит таблицу, которую можно использовать как отправную точку для SVA. В этой таблице выделены семь функциональных областей, которые следует рассматривать: согласованность данных, мертвый или отключенный код, задачи, планирование, память и доступ к устройствам ввода-вывода, очереди, а также прерывания и исключения [26]. Для удобства эта таблица включена в приложение В.

В разделе 5.f FAA AC 20-148 поясняется, что разработчик RSC, например разработчик ОСПВ, должен подготовить следующее:

Анализ поведения RSC, которое может неблагоприятно повлиять на реализацию у пользователей (например, уязвимости, требования к обособлению, последствия аппаратных отказов, требования к резервированию, латентность данных и проектные ограничения для корректной работы RSC). Этот анализ может поддерживать анализ безопасности, выполняемый интегратором или заявителем [27].

Анализ уязвимостей ПО (SVA) по сути аналогичен оценке безопасности функциональности ОСПВ.[†] Для ОСПВ с обособлением SVA обычно объединяют с анализом обособления, поскольку многие уязвимости относятся именно к области обособления. Однако подход меняется от проекта к проекту; стандартизованной схемы оформления этого анализа не существует. Некоторые разработчики ОСПВ оформляют его как SVA, некоторые как анализ опасностей, а другие как анализ обособления.

20.7.2 Вопросы сертификации, которые следует учитывать

В этом разделе перечислены некоторые распространенные сценарии или проблемы, связанные с сертификацией, которые встречаются при использовании ОСПВ в авиационной системе, критичной по безопасности.

20.7.2.1 Формирование безопасного подмножества

Многие из современных ОСРВ, применяемых в приложениях, критичных по безопасности, основаны на коммерчески доступной операционной системе с широким распространением и подтвержденной репутацией по качеству и производительности. Большинство ОСРВ общего назначения разрабатывались с приоритетом *time-to-market* (быстрого вывода на рынок), а не безопасности. Чтобы сделать ОСРВ пригодной для приложений, критичных по безопасности, создают новую ОСРВ, используя подмножество полнофункциональной ОСРВ общего назначения. Любая функциональность, которая может оказаться непригодной для среды, критичной по безопасности, удаляется или изменяется. Выявление проблем безопасности и их устранение либо изменение без влияния на остальную функциональность ОСРВ может быть сложной задачей и должно выполняться с осторожностью. Это требует детального знания проекта ОСРВ и ее кода.

20.7.2.2 Руководство пользователя

Нередко при создании безопасного подмножества руководство пользователя не обновляется в соответствии с этим подмножеством ОСРВ. Это может привести к неправильному пониманию ОСРВ или к ее некорректному использованию.

20.7.2.3 Обратная разработка

Чтобы соответствовать DO-178C (или DO-178B), данные ЖЦ ПО для ОСРВ часто получают путем обратной разработки по исходному коду. В ходе такой работы могут быть выявлены проблемы и в самом исходном коде. Как будет обсуждаться в главе 25, при обратной разработке необходимо учитывать целый ряд вопросов.

20.7.2.4 Отключенные функции

Большинство ОСРВ проектируется для использования множеством заказчиков и содержит функции, которые не обязательно используются или нужны в каждом проекте. Некоторые ОСРВ создаются масштабируемыми, чтобы пользователь мог выбрать, скомпилировать или линковать только те функции ОСРВ, которые ему нужны. У других ОСРВ могут быть функции, которые доступны, но не используются. ОСРВ, создаваемая специально под приложение, имеет то преимущество, что в ней присутствуют только те функции, которые используются приложением. Построение подмножества на основе конкретного приложения может быть затруднено, поскольку функции ОСРВ часто взаимосвязаны. Более распространенный подход состоит в том, чтобы определить и верифицировать ОСРВ независимо от конкретных функций, используемых каждым приложением, а затем рассматривать неиспользуемую функциональность ОСРВ как отключенный код. В главе 17 обсуждается отключенный код и некоторые вопросы, которые следует учитывать, когда функции присутствуют в ОСРВ, но не используются приложениями.

20.7.2.5 Сложность

Некоторые ОСРВ чрезвычайно сложны, содержат код со сложными взаимодействиями и функции, которые могут оказаться ненужными. Такая сложность затрудняет доказательство детерминизма и характеристик безопасности. Кроме того, анализ межкомпонентной связности по данным и по управлению может оказаться затруднен, если сложностью не управлять. Практики проектирования и кодирования, уменьшающие сложность, рассматриваются в главах 7 и 8.

20.7.2.6 Рассогласование с системой

Поскольку ОСРВ разрабатывается отдельно от системы, которая будет ее использовать, между потребностями и ожиданиями системы и функциональностью ОСРВ могут возникать несоответствия. Кроме того, поскольку ОСРВ является частью бортового программного обеспечения, она должна быть увязана с системными требованиями или требованиями к ПО. Система, реализующая ОСРВ, должна иметь требования (обычно требования к ПО), которые трассируются или связываются с требованиями ОСРВ, чтобы гарантировать, что функциональность ОСРВ поддерживает общие потребности системы, выполняет предназначенные функции и не содержит нежелательной или неиспользуемой функциональности.

20.7.2.7 Проблемы соответствия кода

Если ОСРВ изначально не разрабатывалась для среды, критичной по безопасности, и для соответствия DO-178C (или DO-178B), то сам код часто не дотягивает до ожидаемого уровня. Более того, код мог быть создан вообще без каких-либо стандартов кодирования. Нередко код коммерческих ОСРВ содержит мало комментариев и настолько сложен, что даже его исходный автор уже может не понимать, что именно предполагалось. Кроме того, код обычно написан с использованием сложных структур данных, глубокой вложенности и тесной взаимосвязанности, что затрудняет верификацию.

20.7.2.8 Проблемы обработки ошибок

Обработка ошибок – важная функция встроенных систем. ОСРВ часто применяется именно для помощи в обработке ошибок. Любопытно, что обработка ошибок может составлять существенную часть кода ОСРВ. К сожалению, сами обработчики ошибок ОСРВ тоже подвержены ошибкам; причина в том, что они могли быть недостаточно верифицированы, поскольку условия их активации бывают редкими или экзотическими. Нередка ситуация, когда отказы регистрируются на нижних уровнях, но не становятся видимыми на уровне интерфейса прикладного программирования; проблема при этом может заключаться не в самом обработчике ошибок, а в коде, который должен этим обработчиком пользоваться.

20.7.2.9 Регистрация сообщений о проблемах

Поскольку многие ОСРВ разрабатываются организациями, независимыми от разработчика приложения или системного интегратора, сообщения о проблемах (PR), формируемые разработчиком ОСРВ, могут быть недоступны пользователям (например, разработчикам приложений, системным интеграторам, производителям авионики или производителям воздушных судов). В рамках сертификации все сообщения о проблемах (PR) должны быть доступны заявителю на воздушное судно, и по ним должно быть подтверждено отсутствие влияния на безопасность. Это непрерывный процесс. Некоторые разработчики ОСРВ обычно не делятся сообщениями о проблемах со своими заказчиками на протяжении всего срока жизни ОСРВ, иногда просто из опасения, что продукт будет выглядеть плохо. При выборе готовых коммерческих ОСРВ пользователь должен убедиться, что сообщения о проблемах (PR) предоставляются по мере выявления проблем и в течение всего времени, пока ОСРВ используется на воздушном судне.

20.7.2.10 Анализ обособления

Как отмечалось ранее, многие ОСРВ поддерживают надежное обособление. Чтобы доказать достаточность надежного обособления, выполняется анализ обособления. Анализируется ядро ОСРВ, а также поддерживающие компоненты (например, пакет поддержки платы, драйверы устройств, центральный процессор, блок управления памятью и аппаратные устройства), чтобы подтвердить достаточность обособления. Анализ обособления подобен анализу безопасности реализации обособления. В нем рассматривается, каким образом могут возникать нарушения обособления при использовании общих временных, пространственных ресурсов и ресурсов ввода-вывода; затем для каждой уязвимости требуются меры ослабления. В некоторых проектах анализ обособления может оказаться неполным или недостаточным для доказательства надежного обособления. Анализ обособления подробнее рассматривается в главе 21.

20.7.2.11 Другое поддерживающее программное обеспечение

Часто основное внимание сосредоточено на ядре ОСРВ. Однако библиотеки, пакет поддержки платы, промежуточное программное обеспечение, драйверы устройств и т.д., которые связывают ОСРВ с аппаратурой или приложениями, также необходимо учитывать. Иначе говоря, они тоже должны соответствовать DO-178C, поскольку являются частью системы, критичной по безопасности.

20.7.2.12 Испытания на целевом вычислителе

Во многих случаях ОСРВ сначала испытывается с использованием готового коммерческого пакета поддержки платы и готовой коммерческой аппаратуры. Однако ОСРВ должна быть испытана на реальной аппаратуре и с тем поддерживающим программным обеспечением,

которое будет фактически установлено. Обычно для этого требуются дополнительные тесты и модификации некоторых уже существующих тестов, применявшихся с готовым коммерческим пакетом поддержки платы и готовой коммерческой аппаратурой. Испытания на целевом вычислителе важны не только для подтверждения функциональности ОСПВ, пакета поддержки платы и драйверов устройств, но и для проверки того, что готовый коммерческий процессор, блок управления памятью и т.д. работают так, как предполагается. Одним из ключевых критериев при выборе ОСПВ является наличие наборов тестов и ожидаемый объем доработки, необходимый для повторного использования этих тестов для сертификационного зачета.

20.7.2.13 Модификации

Хотя повторное использование ОСПВ является желаемой целью, полностью она реализуется редко. АС 20-148 содержит руководство по RSC. Однако в реальности большинство ОСПВ модифицируется под каждого конкретного пользователя. Важно понимать, что именно изменяется, а что нет (то есть необходим тщательный анализ влияния изменений). Обычно рекомендуется заново испытывать всю ОСПВ при модификации под разных пользователей или при интеграции с различной аппаратурой. Автоматизированные наборы тестов обычно делают эту задачу относительно прямолинейной.

20.8 Другие темы, связанные с ОСПВ

В этом разделе рассматриваются различные темы, связанные с ОСПВ, включая обзор ARINC 653, инструментальную поддержку, ОСПВ с открытым исходным кодом, многоядерные процессоры, виртуализацию и технологию гипервизоров.

20.8.1 Обзор ARINC 653

ARINC 653, *Avionics Application Software Standard Interface*, – это стандарт из трех частей, который в двусторонней печати занимает около трех дюймов толщины. Здесь приводится только обзор высокого уровня. ARINC 653 определяет интерфейс APEX между приложениями и лежащей ниже системой времени выполнения. Он задает стандарт интерфейса прикладного программирования для операционной системы с надежным обособлением, но при этом допускает гибкость в реализации ОСПВ.

Paul Prisaznuk объясняет два ключевых преимущества стандартизованного интерфейса ОСПВ по ARINC 653. Во-первых, он задает четкую границу интерфейса для разработки авионического программного обеспечения. Авионические программные приложения и лежащее ниже основное ПО могут разрабатываться независимо, что позволяет параллельно разрабатывать ОСПВ и приложения, использующие ОСПВ. Кроме того, одни и те же приложения могут быть переносимы на другие платформы, соответствующие ARINC 653. Во-вторых, определение интерфейса ОСПВ по ARINC 653 позволяет лежащему ниже

основному ПО и аппаратной платформе эволюционировать независимо от программных приложений, которые будут на них размещаться [13].

интерфейс APEX определяет для ОСРВ следующую функциональность: (1) отправка и прием данных от и к другим программным приложениям или сетевым системам; (2) управление такими ресурсами, как время, память, ввод-вывод и дисплеи; (3) обработка ошибок; (4) управление файлами; и (5) планирование программных задач или процессов.

По сравнению с другими интерфейсами прикладного программирования у интерфейса APEX есть ряд характерных особенностей. Во-первых, он разработан специально для авиационного сообщества с учетом требований безопасности. Во-вторых, он рассчитан на вычислительную среду с обособлением. Он включает двухуровневое планирование приложений (между разделами и внутри разделов), чтобы один раздел не влиял на другой. Он также определяет межраздельные коммуникации, позволяющие безопасно обмениваться данными между разделами. В-третьих, он предоставляет интерфейс контроля состояния, через который ошибки и состояния на уровне вычислительного модуля могут доводиться до приложений, а ошибки уровня приложений могут передаваться и обрабатываться на уровне модуля (платы) [3]. В разделе 20.4.2 и таблице 20.1 приводится сравнение на высоком уровне между возможностями APEX в ARINC 653 и возможностями POSIX.

Помимо интерфейса прикладного программирования, APEX предоставляет стандартизованную, настраиваемую операционную среду для программных приложений, способствующую абстрагированию прикладного программного обеспечения от лежащей ниже аппаратуры. ARINC 653 определяет спецификацию конфигурации и предполагаемую операционную среду.

Базовая идея ARINC 653 состоит в том, что приложению выделяется раздел (что-то вроде контейнера), внутри которого оно функционирует. Параметры раздела определяются такими атрибутами, как объем памяти, использование процессора (например, в терминах периода и длительности), межраздельные порты и соединения ввода-вывода (каналы связи), а также действия контроля состояния. Атрибуты раздела определяются конфигурационными данными, задаваемыми системным интегратором.

ARINC 653 состоит из трех частей. Каждая из них кратко описана ниже.

Часть 1 определяет обязательные службы, необходимые для удовлетворения требований безопасности, обеспечения переносимости приложений и организации связи между разделами. В части 1 определены следующие базовые службы [3]:

- *Управление разделами*: позволяет приложениям (даже с различными уровнями критичности) выполняться на одной и той же аппаратуре без недопустимого влияния друг на

друга ни по пространству, ни по времени.

- *Управление и контроль процессов*: включает ресурсы, необходимые для управления процессами внутри одного раздела.
- *Управление временем*: позволяет управлять одним из наиболее критичных ресурсов, контролируемых операционной системой, то есть временем. В философии АРЕХ время не зависит от выполнения процесса или раздела. Все значения времени соотносятся со временем основного модуля, а не являются относительными по отношению к разделу или процессу.
- *Механизмы межраздельной связи*: позволяют обмениваться сообщениями между двумя или более разделами, выполняющимися либо на одной и той же аппаратуре, либо на разной аппаратуре. Предусмотрены две парадигмы: очереди для сообщений переменного размера и выборка для сообщений фиксированного размера.
- *Механизмы внутрираздельной связи*: позволяют обмениваться сообщениями между процессами внутри одного и того же раздела без накладных расходов, необходимых для глобальной передачи сообщений.
- *Функции контроля состояния*: отслеживают и сообщают о неисправностях и отказах платформы и приложений. Они также помогают локализовать неисправности, чтобы предотвратить распространение отказов. В системе интегрированной модульной авионики монитор состояния выполняет контроль неисправностей, локализацию неисправностей и управление неисправностями как на уровне приложений, так и на уровне системы.

Часть 2 расширяет службы части 1, добавляя следующее [28]:

1. *Файловая система*: универсальный абстрактный способ управления хранением данных, как отмечалось в разделе 20.6.8.
2. *Структуры данных выборочных портов*: стандартизованный набор структур данных для обмена параметрическими данными. Они помогают уменьшить ненужную вариативность форматов параметров и тем самым сократить потребность в специальной обработке ввода-вывода. Эти структуры данных также обеспечивают переносимость приложений и повышают эффективность основного программного обеспечения.
3. *Несколько расписаний модуля*: расширяют единственное статическое расписание модуля из части 1, позволяя определять несколько расписаний в таблице конфигурации.
4. *Журнальная система*: используется для хранения сообщений. Она сохраняет записанные данные после отказа питания, так что данные можно восстановить после восстановления питания модуля. Каждый журнал доступен только одному разделу, а содер-

жимое и состояние журнала не изменяются при сбросе раздела.

5. *Расширения выборочных портов*: расширяют базовые службы части 1 для выборочных портов следующими службами:
 - a. READ_UPDATED_SAMPLING_MESSAGE
 - b. GET_SAMPLING_PORT_CURRENT_STATUS
 - c. READ_SAMPLING_MESSAGE_CONDITIONAL
6. *Порт доступа для поддержки (SAP)*: особый вид очередного порта, который позволяет получать доступ к адресной информации при отправке и приеме сообщений.
7. *Служба имен*: дополняет службы SAP. Она позволяет разделу получить адрес по имени и получить имя по адресу.
8. *Блоки памяти*: предоставляют разделу способ доступа к блокам памяти в адресном пространстве памяти модуля. Права доступа для раздела определяются в таблицах конфигурации. Разделам может быть предоставлен доступ к блоку памяти только для чтения или для чтения и записи.

Часть 3 представляет собой спецификацию испытаний на соответствие. Она описывает тестовые утверждения и отклики, необходимые для демонстрации соответствия интерфейсу программных служб, определенному в части 1. Эта спецификация предназначена для оценки соответствия ARINC 653.

Большинство операционных систем реального времени соответствуют некоторым частям ARINC 653 части 1 и части 2, но не всем. На данный момент часть 3 не требуется для сертификации, поскольку интерфейсы прикладного программирования испытываются в рамках соответствия DO-178B или DO-178C. В будущем она может использоваться потенциальными заказчиками операционных систем реального времени для оценки их функциональности и подтверждения соответствия ARINC 653. Кроме того, если администрирование ARINC 653 перейдет к независимому органу по стандартизации, испытания на соответствие станут ценным показателем соблюдения стандарта.

ARINC 653 регулярно обновляется в ответ на потребности авиационного сообщества; поэтому важно подтверждать как используемую версию ARINC 653, так и конкретные реализованные службы.

20.8.2 Инструментальная поддержка

Для успешной интеграции и верификации установленной операционной системы реального времени, а также для анализа приложений реального времени, использующих такую систему, необходима существенная инструментальная поддержка. Обычно операционная систе-

ма реального времени сопровождается интегрированной средой разработки. Gerardo Garcia пишет: «Среда разработки оказывает огромное влияние на качество и скорость разработки, а также на общий успех проекта» [18].

Предоставляются инструменты для поддержки разработки и анализа кода. К инструментам разработки кода относятся редактор, ассемблер, компилятор, отладчик, браузер кода и т.д. К инструментам анализа во время выполнения относятся отладчик, анализатор покрытия, монитор производительности и т.д. «Отладочные возможности ваших инструментов разработки ПО реального времени могут означать разницу между созданием успешной системы управления и уходом в бесконечный поиск неуловимых ошибок» [18]. Отладочные инструменты, такие как профилировщики приложений, средства анализа памяти и средства трассировки выполнения, помогают оптимизировать приложения реального времени [18].

Также предоставляются инструменты для оценки поведения операционной системы реального времени и производительности приложений. К таким инструментам относятся средства анализа временного поведения задач и процедур обслуживания прерываний, эффектов, вызванных взаимодействием задача/задача и задача/процедура обслуживания прерываний, таких как синхронизация и вытеснение, проблем, связанных с механизмами защиты ресурсов, например инверсии приоритетов и взаимных блокировок, а также накладных расходов и задержек из-за межзадачного взаимодействия [4].

Кроме того, часто предоставляются инструменты, помогающие конфигурировать операционную систему реального времени. Инструменты конфигурирования помогают настраивать память, ограничения обособления, механизмы связи, например буферы и порты, устройства ввода-вывода, параметры контроля состояния и т.д. Для платформ, соответствующих ARINC 653, многие детали развертывания и реализации определяются в таблицах конфигурации. Некоторые разработчики операционных систем реального времени или платформ предоставляют инструменты для поддержки конфигурирования; однако эта область все еще развивается. Horváth и соавторы утверждают, что несмотря на сложность конфигураций ARINC 653, существующие инструменты доступны только для проектирования на очень низком уровне. На более высоких уровнях не хватает инструментов для фиксации процесса конфигурирования, проверки проектных ограничений конфигурации, регистрации проектных решений, трассируемости конфигурационных данных к требованиям и т.д. «В результате верификация таблиц конфигурации является утомительной работой» [29].

20.8.3 Операционные системы реального времени с открытым исходным кодом

Операционная система реального времени с открытым исходным кодом – это такая операционная система реального времени, исходный код которой общедоступен и бесплатно предоставляется для использования и/или модификации по отношению к исходному про-

екту. Такие системы обычно создаются как совместное усилие, в рамках которого программисты улучшают код и делятся изменениями внутри сообщества. Существует ряд операционных систем реального времени с открытым исходным кодом, например Linux, который привлек значительное внимание. Однако на сегодняшний день, насколько мне известно, не существует операционной системы реального времени с открытым исходным кодом, для которой были бы доступны сопутствующие данные DO-178C, необходимые для поддержки сертификации. Кроме того, код регулярно обновляется, обычно без учета влияния на безопасность. Если использовать такую операционную систему с открытым исходным кодом в системе, критичной по безопасности, то, вероятно, пришлось бы либо реализовать архитектурное смягчение, например обертки, чтобы ограничить влияние операционной системы реального времени, либо восстановить данные ЖЦ ПО по DO-178C методом обратной разработки, используя зафиксированную базовую версию кода. Выполняются некоторые финансируемые государством исследования, чтобы выяснить, можно ли получить преимущества открытого исходного кода и огромного набора доступных инструментов, одновременно соблюдая требования безопасности и сертификации.

Serge Goiffon и Pierre Gaufillet провели исследование, рассматривавшее ARINC 653 и Linux. В их статье указывается, что Linux не готов к соответствию DO-178B (а теперь и DO-178C) по следующим причинам: отсутствуют планы разработки и верификации, среда разработки неоднородна и сложна, распределена через Интернет, многоплатформенна и т.д., не используются универсальные стандарты разработки, то есть стандарты требований, проектирования или кода, и практически отсутствует проектная документация [8]. В статье также были определены некоторые шаги, необходимые для решения задач сертификации по DO-178B[*], включая обратную разработку недостающих данных, добавление планирования разделов ARINC 653 и обеспечение соответствия интерфейсу APEX [8]. Подготовка операционной системы реального времени с открытым исходным кодом к использованию в приложении, критичном по безопасности и требующем DO-178C, была бы крупным мероприятием.

20.8.4 Многоядерные процессоры, виртуализация и гипервизоры

Многие авионические организации рассматривают реализуемость многоядерных процессоров и использование технологий виртуализации и гипервизоров. *Виртуализация* предоставляет программную среду, в которой программы, включая целые операционные системы, могут работать так, как если бы они выполнялись непосредственно на аппаратуре, хотя в действительности они работают не на аппаратуре, а на промежуточном слое, называемом *виртуальной машиной*. Виртуальная машина по сути является изолированным дубликатом реальной машины. Программный слой обеспечивает среду виртуальной машины и обычно называется *монитором виртуальных машин (virtual machine monitor, VMM)* или *гипервизо-*

ром. Монитор виртуальных машин обладает следующими существенными характеристиками [30]:

- Он предоставляет среду, которая выглядит идентичной исходной машине.
- Программы, выполняющиеся в этой среде, работают лишь с незначительным снижением скорости. Монитор виртуальных машин полностью управляет системными ресурсами.

Виртуализация получила популярность в широком вычислительном сообществе и теперь начинает предлагаться вместе с готовыми коммерческими операционными системами реального времени. Она изолирует операционные системы, приложения, устройства и данные друг от друга, пока они работают на одной и той же аппаратной платформе. Для предоставления служб виртуализации и защиты используется гипервизор. Чтобы настроить производительность, размер гипервизора делают относительно небольшим [30].

Если используется виртуализация или технология гипервизора, следует учитывать DO-332, *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*. К такой технологии может относиться часть DO-332, посвященная связанным техникам.

На данный момент использование многоядерных процессоров, виртуализации и гипервизоров рассматривается для установки на воздушные суда. Однако, насколько мне известно, ни одно из этих решений еще не одобрено для установки на коммерческое гражданское воздушное судно. Производители и сертифицирующие органы в настоящее время проводят исследования, выявляют проблемы и работают над их устранением. У меня нет сомнений, что такая технология будет использоваться на коммерческих воздушных судах в не слишком отдаленном будущем.

20.8.5 Защищенность

Многие операционные системы реального времени должны соответствовать как стандартам безопасности, так и стандартам защищенности, особенно когда такая система используется в военной авиации. Для области защищенности применяются Общие критерии (Common Criteria)[*]. Эта система имеет семь уровней доверия к оценке, уровень доверия к оценке (Evaluation Assurance Level, EAL), причем уровень 7 является наивысшим. Для операционных систем реального времени, применяемых и в области безопасности, и в области защищенности, используются как DO-178C, так и Общие критерии.[†]

20.8.6 Вопросы выбора операционной системы реального времени

При выборе операционной системы реального времени для использования в системах, критичных по безопасности, необходимо оценивать несколько аспектов. В приложении С приводятся три категории вопросов, которые следует учитывать при таком выборе: (1) общие

вопросы по операционной системе реального времени, (2) вопросы по ее функциональности и (3) вопросы по ее интеграции.

Ссылки

1. H. M. Deitel, *Operating Systems*, 2-е изд. (Reading, MA: Addison-Wesley, 1990).
2. W. Stallings, *Operating Systems Internals and Principles*, 3-е изд. (Upper Saddle River, NJ: Prentice Hall, 1998).
3. Aeronautical Radio, Inc., Avionics application software standard interface part 1-Required services, ARINC Specification 653P1-3 (Annapolis, MD: Airlines Electronic Engineering Committee, November 2010).
4. J. Cooling, *Software Engineering for Real-Time Systems* (Harlow, U.K.: Addison-Wesley, 2003).
5. I. Bate and P. Conmy, Safe composition of real time software, труды Ninth IEEE International Symposium on High-Assurance Systems Engineering (Dallas, TX, 2005).
6. B. L. Di Vito, A formal model of partitioning for integrated modular avionics, NASA/CR-1998-208703 (Hampton, VA: Langley Research Center, August 1998).
7. E. Klein, RTOS design: How is your application affected? Embedded Systems Conference (San Jose, CA, Spring 1999).
8. S. Goiffon and P. Gaufillet, Linux: A multi-purpose executive support for civil avionics applications? *IFIP-International Federation for Information Processing*, 156, 719-724, 2004.
9. J. Krodel, Commercial off-the-shelf real-time operating system and architectural considerations, DOT/FAA/AR-03/77 (Washington, DC: Office of Aviation Research, February 2004).
10. K. Driscoll, Integrated modular avionics (IMA) requirements and development, Integrated Modular Avionics Conference for the European Network of Excellence on Embedded Systems (Rome, Italy, 2007).
11. RTCA DO-297, *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations* (Washington, DC: RTCA, Inc., November 2005).
12. J. Krodel and G. Romanski, Real-time operating systems and component integration considerations in integrated modular avionics systems report, DOT/FAA/AR-07/39 (Washington, DC: Office of Aviation Research, August 2007).
13. P.J. Prisaznuk, ARINC 653 role in integrated modular avionics (IMA), IEEE Digital Avionics Systems Conference (St. Paul, MN, 2008), pp. 1.E.5-1-1.E.5-10.

14. D. Kleidermacher and M. Griglock, Safety-critical operating systems, *Embedded Systems Programming* 14(10), 22-36, September 2001.
15. W. Lamie and J. Carbone, Measure your RTOS's real-time performance, *Embedded Systems Design* 20(5), 44-53, May 2007.
16. R. G. Landman, Selecting a real-time operating system, *Embedded Systems Programming* 79-96, April 1996.
17. N. Lethaby, Reduce RTOS latency in interrupt-intensive apps, *Embedded Systems Design* 23-27, June 2009.
18. G. Garcia, Choose an RTOS, *Embedded Systems Design* 36-41, November 2007.
19. IAR Systems, How to choose an RTOS, Embedded Systems Conference (San Jose, CA, 2011), pp. 1-22.
20. G. Romanski, Certification of an operating system as a reusable component, IEEE Digital Avionics Systems Conference (Irvine, CA, 2002), pp. 1-8.
21. S. Ferzetti, Real time operating systems (RTOS), on-line tutorial. http://www.slidefinder.net/r/real_t (дата обращения: December 2011).
22. P. Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook* (New York: IEEE Press, 1992).
23. K. Renwick and B. Renwick, How to use priority inheritance, *EE Times*, article #4024970, May 2004.
 - <http://www.eetimes.com/General/PrintView/4024970> (дата обращения: December 2011).
24. C. Z. Yang, Embedded RTOS memory management, YZUCSE SYSLAB tutorial. <http://syslab.cse.yzu.edu.tw/~czyang> (дата обращения: December 2011).
25. F. M. Proctor and W. P. Shackelford, Real-time operating system timing jitter and its impact on motor control, труды SPIE Sensors and Controls for Intelligent Manufacturing II (Boston, MA, October 2011), Vol. 4563, pp. 10-16.
26. V. Halwan and J. Krodel, Study of commercial off-the-shelf (COTS) realtime operating systems (RTOS) in aviation applications, DOT/FAA/AR-02/118 (Washington, DC: Office of Aviation Research, December 2002).
27. Federal Aviation Administration, *Reusable Software Components*, Advisory Circular 20-148 (December 2004).
28. Aeronautical Radio, Inc., Avionics application software standard interface part 2-Extended services, ARINC Specification 653P2-1 (Annapolis, MD: Airlines Electronic Engineering

Committee, December 2008).

29. Á. Horváth, D. Varró, and T. Schoofs, Model-driven development of ARINC 653 configuration tables, IEEE Digital Avionics Systems Conference (Salt Lake City, UT, 2010), 6.E.3-1-6.E.3-15.

30. G. Heiser, Virtualizing embedded Linux, *Embedded Systems Design* 18-26, February 2008.

*Эти определения основаны прежде всего на ARINC 653, части 1-3, озаглавленном *Avionics Application Software Standard Interface: Part 1-Required Services* [3].

*На момент написания книги RTOS соответствовали DO-178B, но с публикацией DO-178C они будут переходить на DO-178C. Для RTOS это должно быть относительно простым переходом, поскольку большинство из них не использует объектно-ориентированные технологии, модельно-ориентированную разработку или формальные методы.

†Некоторые части ОСРВ зависят от аппаратуры, например код, выполняющий переключение контекста, однако такой код стараются свести к минимуму.

*Квадратные скобки добавлены для ясности.

†Большинство RTOS, соответствовавших DO-178B, будут соответствовать и DO-178C, если только в них не используются объектно-ориентированные или связанные техники программирования, модельно-ориентированная разработка или формальные методы. Однако может потребоваться уделить некоторое внимание руководству DO-178C по параметрическим данным, межкомпонентной связности по данным и по управлению, структурному покрытию и данным трассировки.

‡FAA сертифицирует воздушные суда, двигатели и воздушные винты, а не отдельные компоненты.

*Для систем без обособления требуется только второй вид планируемости.

†Квадратные скобки добавлены для пояснения. В спецификации ARINC 653 предлагаются два механизма планирования. Module Operating System (MOS) обеспечивает планирование разделов, а Partition Operating System (POS) обеспечивает планирование процессов внутри раздела.

‡Хотя этот документ уже несколько устарел, отчет FAA DOT/FAA/AR-05/27 под названием *Real-time scheduling analysis* содержит полезную информацию о планировании реального времени в авиационной отрасли. Он доступен на сайте FAA: www.faa.gov.

*Это похоже на deadline monotonic scheduling, за исключением того, что earliest deadline first scheduling является динамическим.

*Следует отметить, что накладные расходы на обработку прерываний – не единственные

накладные расходы, снижающие пропускную способность, то есть полезный объем вычислений, выполняемых приложениями за единицу времени. Переключение контекста, периодический встроенный контроль, контроль состояния и т.д. тоже требуют времени и влияют на пропускную способность.

†В части 2 ARINC 653 файловая система включена как расширенная служба.

*Квадратные скобки добавлены для пояснения.

†Bate и Conmy объясняют типичные шаги выполнения анализа отказов RTOS в своей статье *Safe composition of real time software* [5].

*Статья была написана до публикации DO-178C.

- *Common Criteria* относится к ISO/IEC 15408, *Common Criteria for Information Technology Security Evaluation*.

†Ранее уровни EAL присваивались только самой RTOS; теперь же оценка EAL основывается на системе в целом.

Глава 21. Обособление ПО

Сокращения

Сокращение	Расшифровка
AFDX	Коммутируемый полнодуплексный Ethernet авионики
API	Интерфейс прикладного программирования
BSP	Пакет поддержки платы
CAST	Группа сертифицирующих органов по программному обеспечению
COTS	Готовое коммерческое ПО
CPU	Центральный процессор
CRC	Контроль по циклическому избыточному коду
DMA	Прямой доступ к памяти
FAA	Федеральное управление гражданской авиации США
I/O	Ввод-вывод
IMA	Интегрированная модульная авионика
LRU	Линейно заменяемый блок
MMU	Устройство управления памятью
RAM	Оперативная память
ROM	Постоянная память только для чтения
RTOS	ОСРВ (операционная система реального времени)
SFI	Изоляция программных неисправностей
SVA	Анализ уязвимостей ПО

21.1 Введение в обособление

Обособление является ахиллесовой пятой систем интегрированной модульной авионики (IMA) и современной авионики. Реализация системы с надежным обособлением требует предельной аккуратности и осторожности. В этой главе дается обзор базовых понятий, связанных с обособлением, и ключевых действий, необходимых для реализации обособления в системе, критичной по безопасности.

21.1.1 Обособление как подмножество защиты

Обособление является подмножеством более широкого понятия, называемого *защитой*. Концепции защиты описаны в документе группы сертифицирующих органов по программному обеспечению (Certification Authorities Software Team, CAST)[*] CAST-2 под

названием *Guidelines for assessing software partitioning/protection schemes*. В CAST-2 понятия защиты представлены следующим образом [1]:

- *Двунаправленная защита*. Компонент X защищен от компонента Y, и компонент Y защищен от компонента X. Пример двунаправленной защиты – два компонента внутри авионического блока, между которыми нет взаимодействий и нет общих ресурсов.
- *Однонаправленная защита*. Компонент X защищен от компонента Y, но компонент Y не защищен от компонента X. Пример однонаправленной защиты – авионический блок, который может принимать данные от вычислителя управления полетом, но не может передавать данные в вычислитель управления полетом. Программное обеспечение вычислителя управления полетом может влиять на авионический блок, но не наоборот.
- *Строгая защита*. Компонент X строго защищен от компонента Y, если любое поведение компонента Y не оказывает никакого влияния на работу компонента X. Примером такой защиты являются два компонента внутри авионического блока, между которыми нет взаимодействий и нет общих ресурсов. Строгая защита может быть однонаправленной или двунаправленной.
- *Защита по безопасности*. О компоненте X можно сказать, что он безопасно защищен от компонента Y, если любое поведение компонента Y не оказывает влияния на свойства безопасности компонента X. Примером является применение контроля по циклическому избыточному коду (cyclic redundancy check, CRC) для данных, передаваемых по каналу данных без доверенных гарантий, когда единственным важным с точки зрения безопасности свойством является повреждение данных. Потеря данных в этом примере не является свойством безопасности, представляющим интерес. Защита по безопасности требует определить свойства безопасности на основе анализа безопасности или опасностей. Такая защита может быть однонаправленной или двунаправленной.

Защита может реализовываться программными средствами, аппаратными средствами или их комбинацией. К числу примеров реализации защиты относятся кодирование и декодирование, обертки, инструменты, отдельные аппаратные ресурсы и обособление ПО. В этой главе основное внимание уделяется обособлению ПО, поскольку это наиболее распространенный подход, реализуемый программными средствами. Стоит отметить, что во многих программных документах, например DO-178C (KT-178C), термины *защита* и *обособление* используются как взаимозаменяемые.

Эта глава тесно связана с главой 7 о проектировании и главой 20 об операционных системах реального времени (real-time operating systems, RTOS).

21.1.2 DO-178C и обособление

В разделе 2.4.1 DO-178C обособление описывается как “техника, обеспечивающая изоляцию между компонентами ПО для локализации и (или) изоляции отказов и потенциального уменьшения объема работ процесса верификации ПО” [2]. Далее в DO-178C поясняется, что обособление между компонентами ПО может реализовываться либо путем размещения компонентов ПО на разных аппаратных ресурсах, либо путем выполнения более чем одного компонента ПО на одном и том же оборудовании. С учетом современных скоростей и емкости процессоров часто выбирается второй подход. Раздел 2.4.1 DO-178C дает следующие пять указаний, применимых независимо от подхода к обособлению [2]:

- a. Обособленному компоненту ПО не должно быть позволено повреждать код, области ввода-вывода (input/output, I/O) или области хранения данных другого обособленного компонента ПО.
- b. Обособленный компонент ПО должен использовать общие ресурсы процессора только в течение назначенного ему периода выполнения.
- c. Отказы аппаратуры, уникальной для обособленного компонента ПО, не должны вызывать неблагоприятных эффектов в других обособленных компонентах ПО.
- d. Любое программное обеспечение, обеспечивающее обособление, должно иметь тот же или более высокий уровень ПО, чем наивысший уровень, назначенный любому из обособленных компонентов ПО.
- e. Любая аппаратура, обеспечивающая обособление, должна оцениваться в рамках процесса системной оценки безопасности, чтобы убедиться, что она не оказывает неблагоприятного влияния на безопасность.

В DO-178C есть одна цель, в которой обособление упоминается явно: цель 13 таблицы А-4, где сказано: “Подтверждена целостность обособления ПО” [2]. Эта цель ссылается на раздел 6.3.3.f DO-178C, где объясняется, что при рассмотрении и (или) анализе архитектуры ПО необходимо убедиться, что нарушения обособления предотвращаются. Эта цель применяется ко всем уровням ПО, когда используется обособление.

Есть и пять целей DO-178C, косвенно связанных с обособлением: цели 1 – 5 таблицы А-5 все ссылаются, либо как на цель, либо как на мероприятие, на раздел 6.4.3.a DO-178C, где объясняется, что “нарушения обособления ПО” являются типичным видом ошибок, который должен выявляться во время интеграционных испытаний требований аппаратуры и ПО [2]. Это означает, что обособление должно быть охвачено требованиями и подлежит испытаниям. На практике это непросто, поскольку требования к обособлению часто являются *отрицательными*, например: “Один раздел не должен иметь возможности модифици-

ровать память данных другого раздела, если только эта память не была сконфигурирована как общая”. Написать испытание, демонстрирующее один или два примера, нетрудно, но из-за различных вариантов конфигурации бывает сложно показать, что набора испытаний достаточно для верификации требования.

Обособление также напрямую поддерживает безопасность, когда используется для обособления и (или) изоляции программных функций. Оно гарантирует, что менее критичные функции программного обеспечения не будут мешать более критичным функциям и что все функции имеют время, память и ресурсы ввода-вывода, необходимые для выполнения их предназначенной функциональности.

21.1.3 Надежное обособление

До конца 1990-х годов обособление использовалось довольно редко для обособления двух или более уровней ПО, выполняющихся на одном и том же оборудовании. Однако с конца 1990-х годов оно стало гораздо более распространенным приемом в авионике. С ростом вычислительной мощности и появлением операционных систем реального времени, поддерживающих обособление, обособление стало ключевой характеристикой многих современных авионических систем. Надежное обособление является краеугольным камнем интегрированной модульной авионики. В RTCA DO-297 интегрированная модульная авионика определяется так: “Общий набор гибких, повторно используемых и совместимых аппаратных и программных ресурсов, которые при интеграции образуют платформу, предоставляющую службы, спроектированные и верифицированные на соответствие определенному набору требований по безопасности и характеристикам, для размещения приложений, выполняющих функции воздушного судна” [3]. Поскольку такая платформа размещает приложения различных уровней ПО, требуется надежное обособление, чтобы гарантировать наличие у каждого приложения необходимых ресурсов и отсутствие помех со стороны других приложений.

В главе 20 было введено понятие надежного обособления и отмечено, что оно связано с операционными системами реального времени. В этой главе это понятие рассматривается подробнее. Раздел 2.3.3 DO-297 объясняет понятие надежного обособления следующим образом [3]:

Надежное обособление – это средство обеспечения требуемой изоляции независимых функций воздушного судна и приложений, размещенных в общих ресурсах интегрированной модульной авионики, при наличии ошибок проектирования и аппаратных отказов, уникальных для конкретного раздела или связанных с аппаратурой, специфичной для приложения. Если иной отказ может привести к потере надежного обособления, то он должен обнаруживаться и по нему должны предприниматься соответствующие действия. Цель

надежного обособления состоит в обеспечении того же уровня функциональной, пусть и не обязательно физической, изоляции и защиты, что и при федеративной реализации. Это означает, что надежное обособление должно поддерживать совместное сосуществование базового программного обеспечения и приложений, размещенных на процессоре и использующих общие ресурсы, одновременно гарантируя предотвращение несанкционированного или непреднамеренного взаимного влияния... Надежное обособление – это средство обеспечения требуемой изоляции и независимости при любых обстоятельствах, включая аппаратные отказы, ошибки проектирования аппаратуры и ПО или ненормальное поведение, для функций воздушного судна и размещенных приложений, использующих общие ресурсы. Цель надежного обособления состоит в обеспечении уровня функциональной изоляции и независимости, эквивалентного федеративной реализации системы

(то есть когда приложения размещаются по отдельности на отдельных линейно заменяемых блоках (line replaceable units, LRU)). Это означает, что надежное обособление поддерживает совместное сосуществование приложений, использующих общие ресурсы, одновременно гарантируя обнаружение и ослабление любых попыток несанкционированного или непреднамеренного взаимодействия.

Реализация надежного обособления опирается на область компьютерной безопасности, в которой используются понятия потока данных и информации, то есть управление доступом, невмешательство и разделимость, политики целостности, временные каналы, каналы хранения и отказ в обслуживании. Однако, хотя концепции обособления по безопасности и программной защищенности связаны, эти две модели не полностью совпадают, поскольку определяются разными целями [4,5]. В разделе 3.5 DO-297 перечислены следующие характеристики надежного обособления [3]:

- Службы обособления должны обеспечивать достаточное разграничение и изоляцию функций воздушного судна и размещенных приложений, совместно использующих ресурсы платформы. Службы обособления – это службы, предоставляемые платформой и определяющие и поддерживающие независимость и разграничение между разделами. Эти службы гарантируют, что поведение функций или приложений внутри одного раздела не может неприемлемым образом влиять на поведение функций или приложений в любом другом разделе. Эти службы должны предотвращать любые неблагоприятные последствия для воздушного судна, возникающие из-за одновременного необнаруженного повреждения всех разделов функций и приложений, совместно использующих затронутые ресурсы.
- Должна существовать возможность определить в реальном времени, с надлежащим уровнем доверия, что службы обособления работают так, как задано, и в соответствии

с определенным уровнем безопасности.

- Службы обособления не должны опираться на какое-либо требуемое поведение любой функции воздушного судна или любого размещенного приложения. Это означает, что все механизмы защиты, необходимые для установления и поддержания обособления, предоставляются платформой интегрированной модульной авионики.

Если надежное обособление реализовано неправильно, это может привести к многочисленным проблемам, часть которых может иметь последствия для безопасности. Примеры проблем обособления таковы [3,6]:

- Ошибочная запись данных не в те области. Например, неисправный раздел может позволить приложению записывать в область памяти, к которой другой раздел считает себя имеющим исключительный доступ. Либо один раздел может забирать процессорное время у другого приложения.
- Аварийное завершение работы процессора.
- Повреждение ввода-вывода путем ложной передачи выходных данных, которые выглядят как данные от критичной функции.
- Повреждение входных данных до того, как критичная функция их использует.
- Монополизация внутренних путей связи.
- Повреждение общей файловой системы флэш-памяти.
- Внесение временного джиттера при переключении контекста на новый раздел, что изменяет производительность нового раздела.

Надежное обособление распространяется на три базовые подсистемы любой вычислительной платформы: память, центральный процессор (central processing unit, CPU) и подсистему ввода-вывода [7]. Связь между разделами также может рассматриваться как общий ресурс [8], однако обычно это пересекается с другими общими ресурсами, поэтому отдельно здесь не рассматривается. Для каждого общего ресурса, то есть памяти, времени и ввода-вывода, вопросы обособления должны быть учтены в проектировании, реализации и верификации. Далее рассматривается обособление каждого из этих общих ресурсов.

21.2 Общая память (пространственное обособление)

Джон Рашби пишет: “Пространственное обособление должно гарантировать, что программное обеспечение в одном разделе не может изменить ПО или частные данные другого раздела, будь то в памяти или в передаче, а также не может управлять частными устройствами или исполнительными механизмами других разделов” [4]. По сути, пространственное обособление предотвращает повреждение или перезапись функцией из одного

раздела пространства данных функции из другого раздела [9]. Джастин Литлфилд-Лоуилл и Ларри Киннан поясняют, что пространственное обособление гарантирует, что общие системные ресурсы в одном разделе “не потребляются таким образом, который привел бы к отказу в обслуживании для других разделов, которым требуется доступ к тому же ресурсу” [10]. Существуют два распространенных подхода к защите памяти: (1) использование блока управления памятью (memory management unit, MMU) или (2) использование изоляции программных неисправностей (software fault isolation, SFI).

Аппаратно реализованное пространственное обособление является наиболее распространенной формой пространственного обособления. Аппаратный блок управления памятью обычно предоставляется вместе с центральным процессором. Подробности его работы различаются от процессора к процессору. Блок управления памятью обеспечивает, чтобы политики, выраженные в соответствующих таблицах, задавали требуемый контроль доступа к памяти. Поскольку такой блок нередко является готовым коммерческим изделием (commercial off-the-shelf, COTS) и по нему нет данных ЖЦ ПО, для настройки соответствующих таблиц, которые затем использует процессор, применяется операционная система. Правильность работы этого блока должна быть подтверждена в ходе сертификационных работ. В главе 20 объяснялось, что это часто подтверждается во время испытаний ОСРВ. Отчет Федерального управления гражданской авиации США (FAA) под названием *Commercial Off-The-Shelf Real-Time Operating System and Architectural Considerations* [11] дает дополнительные сведения о том, как устройство управления памятью и операционная система реального времени используются для обеспечения надежного обособления.

Если блок управления памятью не используется, альтернативой является изоляция программных неисправностей [4]. В этом подходе в код в каждой точке доступа к памяти добавляются логические проверки. Машинный код в разделе анализируется для определения назначений обращений к памяти и проверки их корректности.

Косвенные обращения к памяти невозможно проверить статически, поэтому в программу добавляются инструкции, проверяющие содержимое адресного регистра во время выполнения непосредственно перед его использованием. Такой метод вносит определенные накладные расходы, поскольку в программу добавляется дополнительный код. Он также требует дополнительных затрат на анализ и сертификацию в каждом проекте. Однако большую часть процедуры проверки можно автоматизировать, а также квалифицировать инструмент или набор инструментов, которые затем могут использоваться в нескольких проектах [9].

В позиции CAST-2 выделяется несколько областей, которые следует учитывать при реализации обособления памяти, например потерю, повреждение или задержку входных либо выходных данных, повреждение внутренних данных, наложения программ, последователь-

ности буферов, взаимодействие с внешними устройствами, дефекты потока управления, влияющие на память, например ошибочные переходы в раздел или защищенную область, и так далее [1].

21.3 Общий центральный процессор (временное обособление)

“Временное обособление должно гарантировать, что на службу, получаемую программным обеспечением в одном разделе от общих ресурсов, не может повлиять ПО в другом разделе. Это включает производительность соответствующего ресурса, а также скорость, задержку, джиттер и длительность запланированного доступа к нему” [4]. Обособление во временной области тесно связано с планируемостью многозадачности, которая рассматривалась в главе 20. ARINC 653 навязывает строгое циклическое планирование разделов по круговой схеме (round-robin), при котором длительности и периоды задаются в конфигурационной таблице. Внутри раздела используются другие планировщики.

Цель временного обособления состоит в том, чтобы функции в одном разделе не нарушали временные характеристики событий в других разделах. Опасения вызывают ситуации, когда один раздел монополизировать центральный процессор, аварийно останавливает систему или выдает инструкцию HALT, что приводит к отказу в обслуживании других разделов. “К другим сценариям, из-за которых раздел может не освободить центральный процессор вовремя, относятся обычные перерасходы расписания, когда определенные значения параметров приводят к тому, что вычисление занимает больше отведенного ему времени, а также неуправляемое выполнение, когда программа застревает в цикле” [4]. Подход к временному обособлению должен учитывать подобные сценарии.

Прерывания процессора используются в системах реального времени для выявления события, которому требуется доступ к процессору. С прерываниями нужно обращаться осторожно, чтобы не подорвать временное обособление. Обычно такие нарушения предотвращают, полностью исключая прерывания, за исключением тиков таймера, используемых для реализации расписания. Однако некоторые стороны хотели бы расширить ARINC 653, чтобы он предоставлял детерминированный способ обработки прерываний [7]. Подробнее прерывания рассматриваются в разделе 21.5.

В позиции CAST-2 выделяются области, которые следует учитывать при реализации временного обособления, например прерывания и запреты прерываний, циклы, выход за пределы кадра по времени, повреждение счетчика или таймера, конвейеризация и кэширование, дефекты потока управления, конфликты по памяти или вводу-выводу, программные ловушки, например деление на ноль, и так далее [1].

21.4 Общий ввод-вывод

Большинство систем имеет несколько портов, устройств и каналов ввода-вывода, например последовательную шину, оконечную систему ARINC 664 для коммутируемого полнодуплексного Ethernet авионики (Avionics Full-Duplex Switched Ethernet, AFDX), программируемую вентильную матрицу (field-programmable gate array, FPGA) или устройство контроля по циклическому избыточному коду. Некоторые устройства выделены конкретному разделу, а другие совместно используются несколькими разделами. Как отмечалось в главе 20 об ОСПВ, драйвер устройства обычно служит связующим кодом между устройством и ядром ОСПВ.

Как и в случае с другими общими ресурсами, ресурсы ввода-вывода должны быть обособлены. Вопросы обособления для общего ввода-вывода тесно связаны с пространственным и временным обособлением. Для каждого устройства ввода-вывода необходимо учитывать обособление как во временной, так и в пространственной области.

Решение вопросов обособления во вводе-выводе может быть одним из самых сложных аспектов проектирования системы с обособлением. ARINC 653 определяет интерфейсы и операции выборочных и очередных портов для межраздельного ввода-вывода. Однако ввод-вывод к физическим устройствам или межмодульный ввод-вывод реализуются по усмотрению разработчиков ОСПВ или других заинтересованных сторон. Механизм портов ARINC 653, основанный на *псевдопортах* и *псевдоразделах*, может использоваться как стандарт интерфейса для обеспечения связности, однако он способен вызывать проблемы производительности приложений [10]. Стив ВандерЛист объясняет, что при обособлении устройств ввода-вывода необходимо учитывать не только конечное устройство, но и механизмы связи, соединяющие устройство с памятью и процессором, например коммуникационные шины, механизмы прямого доступа к памяти и промежуточные буферы. “Среда обособления должна управлять всеми существенными характеристиками подсистемы ввода-вывода”, включая задержку, пропускную способность, управляющие регистры и буферное пространство, чтобы разделы не могли влиять друг на друга несанкционированным способом [7].

Подход к решению вопросов обособления ввода-вывода меняется в зависимости от особенностей аппаратуры ввода-вывода и низкоуровневого драйвера устройства. В исследовательском отчете Федерального управления гражданской авиации США под названием *Real-Time Operating Systems and Component Integration Considerations in Integrated Modular Avionics Systems Report* отмечается:

Возможны различные подходы, например централизованное управление вводом-выводом в ядре или управление вводом-выводом на уровне раздела. Во многих реализациях используется своего рода таблица разрешений разделов или задач, управляемая ОСПВ, которая

позволяет только определенным разделам или задачам обращаться к конкретному вводу-выводу. Затем применяется монитор состояния для выявления любых неразрешенных обращений со стороны других задач или разделов. Вопросы ввода-вывода различаются для разных ОСРВ [8].

В зависимости от сценария ОСРВ может реализовывать определенные ограничения или делать некоторые предположения, которые требуют от интегратора специальных шагов интеграции [8]. Как и любые другие предположения или ограничения, они должны документироваться поставщиком ОСРВ и (или) поставщиком платформы и доводиться до интегратора. Нередко информация такого рода включается в паспорт платформы.

Использование прерываний ввода-вывода должно тщательно оцениваться в схеме обособления. Большинство процессоров поддерживает прерывания от событий, связанных с вводом-выводом. Однако, как будет показано далее, прерывания могут влиять на обособление и потому могут быть недопустимы. Следовательно, может потребоваться иной способ передачи событий, требующих реакции.

21.5 Некоторые сложности, связанные с обособлением

Некоторые организации склонны полагаться только на ОСРВ для реализации обособления. Однако обособление – это характеристика системного уровня. Процессор, пакет поддержки платы (board support package, BSP), устройства, драйверы устройств, блок управления памятью и так далее участвуют в выбранном подходе к обособлению и могут нарушить надежное временное и пространственное обособление. В этом разделе выделены некоторые конкретные области, создающие сложности для обособления, включая прямой доступ к памяти, кэш-память, прерывания и межраздельное взаимодействие.

21.5.1 Прямой доступ к памяти

Прямой доступ к памяти (direct memory access, DMA) используется для передачи большого блока памяти за короткое время независимо от процессора. Такие передачи используют специальный механизм, который может получать исключительный доступ к шине памяти для выполнения блочной передачи. Шина памяти является общим ресурсом, поэтому при неправильном совместном использовании она может лишить приложение необходимых ему ресурсов. Прямой доступ к памяти может нарушить временное обособление, если передача инициирована разделом, у которого осталось меньше времени выполнения, чем требуется для ее завершения. Он также может вызывать нарушения обособления памяти. Один из способов решить эту проблему состоит в создании интерфейса прикладного программирования (application programming interface, API), управляющего доступом к прямому доступу к памяти, вместо предоставления приложению прямого доступа. Из-за такого интерфейса производительность может пострадать, но этот подход все равно быстрее других

типов передачи памяти, если только размер передаваемых блоков памяти достаточно велик [10].

21.5.2 Кэш-память

Кэш-память – это промежуточная высокоскоростная память, расположенная между основной памятью и центральным процессором. В ней хранится копия часто используемых данных памяти для быстрого доступа. Она существенно повышает производительность. Однако готовые коммерческие процессоры не предоставляют обособленный кэш, выделенный конкретным разделам, хотя ситуация меняется с некоторыми новыми устройствами, планируемыми к выпуску в будущем. В системе с обособлением состояние процессора заменяется при каждом переключении разделов. Состояние до переключения хранится в регистрах процессора. Большинство процессоров предоставляет средства для быстрого выполнения такой замены. В системе без обособления состояние кэш-памяти сохранять не требуется, поскольку имеется одно приложение, которое не мешает самому себе. Однако в системе с обособлением кэш является общим ресурсом [7].

Существует несколько вариантов сохранения обособления при использовании кэш-памяти. Один из вариантов устранения потенциальных нарушений обособления, вызываемых кэшем, – полностью отключить кэш. Однако влияние на производительность обычно слишком велико, поэтому необходимо реализовать детерминированный способ использования кэша. На сегодняшний день наиболее распространенным решением является очистка кэша. Подход состоит в том, чтобы очищать кэш при переключении раздела, чтобы новый раздел начинал работу с чистой кэш-памятью. Как отмечалось ранее, “Под очисткой понимается копирование всех значений, присутствующих только в кэше, обратно в основную память, то есть значений, которые были обновлены и для которых используется режим *copy-back*. Тем самым накладные расходы переносятся на начало выполнения раздела, а не распределяются по всему его времени работы” [11]. Очистка увеличивает время переключения контекста и снижает производительность кэша в начале выполнения раздела, пока кэш заполняется, но это все равно эффективнее, чем совсем обходиться без кэша. Поскольку время, требуемое для операций очистки, зависит от объема данных, записанных в память, необходимо учитывать наихудшее время переключения контекста, чтобы убедиться в соблюдении временных требований. Поскольку каждый временной слот раздела начинается с пустого кэша, производительность приложения может ухудшиться, если длительность раздела слишком мала и кэш просто не успевает как следует заполниться.

Режим сквозной записи кэша (*cache write-through*) является еще одним вариантом решения проблем обособления, связанных с кэшем. Это эффективнее, чем вариант без кэша, однако выполнение кода происходит медленнее. Преимущество этого варианта состоит в том, что

инвалидация кэша выполняется одной быстрой инструкцией.

21.5.3 Прерывания

Прерывание – это сигнал, запускаемый асинхронным событием. Когда прерывания разрешены процессором, обычное выполнение приостанавливается для обслуживания события прерывания с помощью процедуры обслуживания прерывания. Чтобы предотвращать такие нарушения, в системах с надежным обособлением иногда вообще отказываются от прерываний, за исключением тиков таймера, используемых для реализации расписания [10]. ОСРВ с обособлением, соответствующие ARINC 653, как правило, ограничивают прерывания системными часами. Однако для предотвращения нарушений временного обособления во время прерываний могут применяться некоторые специальные аппаратные и (или) программные приемы, например запрет приложению доступа к аппаратуре, способной вызывать прерывания, реализация программного опроса данных, связанных с сигналом прерывания, или реализация интерфейса прикладного программирования, проверяющего, может ли запрошенная операция завершиться до конца текущего малого кадра времени [10]. Такие приемы необходимо очень тщательно анализировать на предмет эффективности, и это, мягко говоря, не всегда тривиальная работа.

21.5.4 Межраздельное взаимодействие

Если бы каждый раздел представлял собой изолированный домен и не нуждался во взаимодействии с другими такими доменами, жить было бы проще. Однако поскольку разделам может требоваться обмениваться информацией друг с другом, межраздельное взаимодействие должно проектироваться так, чтобы поддерживать надежное обособление.

Сложность состоит в том, чтобы спроектировать решение по обособлению, которое позволяет обмен информацией между обособленными функциями, то есть межраздельное взаимодействие, и управляет доступом к другим общим ресурсам, например устройствам ввода-вывода, сохраняя при этом значительную автономность обособленных функций и их невосприимчивость к влиянию других функций. Межраздельное взаимодействие и совместное использование устройств ввода-вывода влияют как на пространственный, так и на временной аспекты обособления и механизмов защиты [9].

При решении задач межраздельного взаимодействия необходимо учитывать как измерение памяти, главным образом связанное с передачей данных из одного раздела в другой, так и временное измерение, в основном касающееся синхронизации и того, как один раздел вызывает службы другого [4]. Связь между разделами должна быть ограничена только теми взаимодействиями, которые предусмотрены и разрешены системными конфигурационными данными. Исследовательский отчет Федерального управления гражданской авиации США содержит рекомендации по реализации межраздельного взаимодействия без наруше-

ния временного или пространственного обособления [9].

21.6 Рекомендации по обособлению

В этом разделе приводятся некоторые практические рекомендации, которые следует учитывать при реализации и верификации обособления. Каждый проект отличается от других, и технические детали необходимо рассматривать применительно к конкретному случаю. Однако эти рекомендации задуманы как отправная точка.

Рекомендация 1: помнить, что надежное обособление является задачей системного уровня. Как пояснялось в разделе 21.5, надежное обособление не реализуется одним только программным обеспечением. Это задача системного уровня, требующая раннего и постоянного взаимодействия между командами системной инженерии, аппаратуры и ПО. Точно так же надежное обособление не реализуется одной лишь ОСРВ.

ОСРВ может играть значительную роль в обеспечении надежного обособления, но это не единственный задействованный компонент.

Рекомендация 2: проектировать надежное обособление проактивно. Надежное обособление требует тщательного проектирования – само по себе оно не возникает. Необходимо учитывать множество измерений и решать многочисленные технические задачи. Однако проактивное решение этих задач на этапах разработки существенно уменьшает количество неприятных сюрпризов на этапах интеграции и верификации. Предлагаются следующие рекомендации:

1. Учитывайте отмеченные ранее типовые проблемы, например прерывания, кэш-память и сложности ввода-вывода, и предусматривайте соответствующие решения в проектном решении.
2. Документируйте цели обособления, например:
 - a. *Система будет реализовывать надежную защиту памяти:* приложение всегда будет получать выделенные ему ресурсы памяти без помех со стороны других приложений.
 - b. *Система будет реализовывать надежную защиту по времени:* приложение всегда будет получать выделенное ему время выполнения без помех со стороны других приложений.
 - c. *Система будет реализовывать надежную защиту ресурсов:* приложение всегда будет получать выделенные ему физические и логические ресурсы без помех со стороны других приложений.
3. Определяйте требования к надежному обособлению, обеспечивающие достижение

этих целей.

4. Выявляйте все общие ресурсы, чтобы их можно было корректно использовать для поддержки надежного обособления.
5. Документируйте детали обособления в проекте. В частности, обеспечьте четкую идентификацию всех компонентов и их взаимодействий.
6. Выявляйте все требования на каждом иерархическом уровне, которые поддерживают обособление или влияют на него. Например, можно назначить атрибут требования, чтобы отличать требования, связанные с обособлением. Это также помогает при последующем анализе обособления.
7. Выявляйте несколько способов предотвращения распространения отказов.
8. Используйте границы локализации, чтобы ограничить последствия отказов.
9. Обращайтесь к руководству раздела 3.5.1 DO-297 под названием *Design for robust partitioning* [3], где приводятся рекомендации, которые следует учитывать при проектировании платформы интегрированной модульной авионики с надежным обособлением. Понятия DO-297 применимы, даже если система формально не классифицируется как интегрированная модульная авионика.

Рекомендация 3: в ходе проектирования и проектных рассмотрений следить за тем, чтобы уязвимости были учтены. По мере документирования и рассмотрения проекта следует задавать проверочные вопросы, которые могли бы выявить нарушения обособления. В таблице 21.1 приведены примеры вопросов, помогающих выявить уязвимости, связанные с данными и управлением.

Таблица 21.1 Примеры вопросов, помогающих выявить уязвимости обособления

Уязвимости, связанные с данными	Уязвимости, связанные с управлением
Может ли обособление быть нарушено потоком данных?	Может ли обособление быть нарушено потоком управления?
Могут ли общие данные использоваться ненадлежащим образом?	Могут ли функции вызываться ненадлежащим образом в пределах одной функциональной возможности или между функциональными возможностями?
Могут ли сообщения отправляться или приниматься некорректно?	Могут ли прерывания вызывать ошибочное поведение?
Могут ли параметры функций использоваться ненадлежащим образом?	Могут ли аппаратные отказы или неисправности влиять на целостность данных или порядок выполнения?
Могут ли конфигурационные данные быть недействительными?	Могут ли переходы между режимами быть реализованы некорректно?
Могут ли данные передаваться некорректно?	Могут ли ресурсы выделяться ненадлежащим образом?
Могут ли данные инициализироваться ненадлежащим образом?	Может ли отключенный код быть непреднамеренно активирован?
Могут ли глобальные данные читаться или записываться ненадлежащим образом?	Может ли последовательность инициализации быть некорректной?
Могут ли глобальные данные ошибочно записываться непредназначенными для этого функциями?	Возможны ли ненадлежащие реакции на исключения?
Могут ли глобальные данные оставаться неинициализированными или повторно инициализироваться ненадлежащим образом?	Могут ли обработчики отказов вести себя ненадлежащим образом, например пропускать отказы или неисправности либо обрабатывать их некорректно?

Уязвимости, связанные с данными	Уязвимости, связанные с управлением
Могут ли аппаратные регистры использоваться ненадлежащим образом?	Могут ли возникать перекрытия областей памяти?
Может ли компоновщик некорректно скомпоновать данные или код?	Возможно ли чтение по неверным аппаратным адресам или запись по ним?
Могут ли данные устаревать или становиться недействительными?	Возможны ли ненадлежащие реакции во время сбросов?
Могут ли данные пропадать?	Может ли синхронизация нарушаться из-за несовпадений или ошибочных аппаратных ожиданий?
Возможны ли некорректные реакции на несовпадения данных?	Может ли некорректное переключение контекста вызывать ошибочные данные или временные нарушения?
Возможны ли неожиданные значения с плавающей точкой?	Могут ли возникать какие-либо неожиданные исключения?

- Могут ли функции выполняться с неверной частотой или в неверные моменты времени?

Рекомендация 4: рассмотреть использование OCPB, соответствующей ARINC 653. Как отмечалось ранее, одна только OCPB не решает все вопросы обособления. Однако это хорошая отправная точка. ARINC 653 дает рекомендации, помогающие разработчикам OCPB и платформ продумать сложности обособления.

Рекомендация 5: использовать другие виды работ по разработке и верификации, чтобы избежать дублирования усилий. Обособление следует учитывать на протяжении всей разработки системы и программного обеспечения. Для поддержки проектирования и верификации надежного обособления можно использовать и другие работы по разработке или верификации, включая следующее:

- *Опирайтесь на работы по анализу межкомпонентной связности по данным и по управлению.* Анализ межкомпонентной связности по данным и по управлению дает хорошую отправную точку для надежного обособления. Межкомпонентная связность по данным и по управлению и обособление ПО связаны между собой, но это разные понятия. Обособление является способом обеспечить изоляцию между независимыми компонентами ПО и тем самым защитить от непреднамеренной связности между независимыми обособленными компонентами. Межкомпонентная связность по данным и по управлению – это преднамеренное взаимодействие между компонентами, включая межкомпонентную связность между отдельно обособленными компонентами. Обособление ПО не гарантирует отсутствие проблем межкомпонентной связности по данным или по управлению; и наоборот, проблемы межкомпонентной связности по данным или по управлению не означают, что механизм обособления ошибочен. Для систем с обособлением требуются как анализы межкомпонентной связности по данным и по управлению, так и анализ обособления. Между этими двумя задачами может быть определенная синергия.
- *Используйте испытания на робастность для подтверждения того, что обособление*

не нарушается. Когда обособление четко задокументировано в требованиях и проекте, это направляет испытания и дает хорошую основу для подтверждения заявлений о надежном обособлении. Анализ обособления может выявить необходимость дополнительных испытаний. Однако тесты, основанные на требованиях, создают фундамент, с которого можно начинать наращивать уверенность в стратегии обособления.

- *Используйте Анализ уязвимостей ПО (Software Vulnerability Analysis, SVA) для ОСРВ.* Если используется ОСРВ, у поставщика ОСРВ может иметься такой анализ, который можно использовать как входные данные для анализа обособления. Этот анализ обсуждался в главе 20.

Рекомендация 6: испытывать механизм обособления. Основная цель верификации механизма обособления состоит в том, чтобы убедиться, что никакое ошибочное поведение в одном разделе не может способствовать некорректному поведению или отказу любого другого раздела, а также в том, что обособление общих ресурсов не нарушается. В информационном документе DO-248C #14 под названием *Partitioning aspects in DO-178C/DO-278A* приводятся некоторые рекомендации по верификации механизма обособления:

Элементы целостности обособления могут быть верифицированы путем проверки механизмов обособления с использованием специальных сценариев испытаний, моделирования и (или) методов анализа. Сценарии испытаний следует писать так, чтобы возбуждать механизм обособления путем внесения ошибок или попыток нарушения для обхода временных и пространственных ограничений. Примером дополнительного анализа может служить расчет наихудшего времени выполнения для оценки временных характеристик... Следует создать набор верификационных испытаний, содержащий нормальные тестовые примеры и ненормальные либо выходящие за допустимый диапазон тестовые примеры для всех требований к механизму обособления. Робастность механизма обособления может быть продемонстрирована с использованием набора испытаний, основанного на требованиях и удовлетворяющего покрытию тестами, основанными на требованиях [12].

В некоторых проектах также выполняются *испытания с разделом-нарушителем*, когда специально разрабатывается раздел, пытающийся преднамеренно нарушить обособление.

Рекомендация 7: выполнять анализ обособления. DO-297 предусматривает мероприятие, называемое *анализом обособления*. Цель этого анализа – продемонстрировать, что никакое приложение в одном разделе не может неблагоприятным образом повлиять на поведение приложений в любом другом разделе. Анализ обособления похож на системную оценку безопасности, при которой рассматриваются и смягчаются все потенциальные источники отказов. Все уязвимости должны быть выявлены, классифицированы и смягчены [11]. Инженеры, выполняющие анализ, должны детально понимать архитектуру системы, аппара-

туры и ПО. Главный архитектор нередко является идеальным кандидатом для выполнения такого анализа. Ниже приведены типовые задачи, выполняемые в рамках анализа обособления:

1. *Собрать данные, поддерживающие анализ*, включая предварительную системную оценку безопасности, системные требования и проектные решения, требования к ПО и проект ПО, аппаратную архитектуру, данные проекта пакета поддержки платы и драйверов устройств, техническое описание процессора и (или) руководство пользователя, руководства пользователя устройств, спецификации интерфейсов, требования и проект конфигурационного инструмента, если применимо, и так далее.

2. *Определить заявления о надежном обособлении*, которые будут анализироваться, например:[*]

- a. Система будет реализовывать надежную защиту памяти: приложение всегда будет получать выделенные ему ресурсы памяти без помех со стороны других приложений.
- b. Система будет реализовывать надежную защиту по времени: приложение всегда будет получать выделенное ему время выполнения без помех со стороны других приложений.
- c. Система будет реализовывать надежную защиту ресурсов: приложение всегда будет получать выделенные ему физические и логические ресурсы без помех со стороны других приложений.

3. *Выявить потенциальные уязвимости, которые могут нарушить каждое заявление.*

Все потенциальные источники ошибок должны выявляться и рассматриваться систематически, включая ограничения ресурсов, задачи планирования, ввод-вывод, источники ошибок прерываний и так далее. Анализ трассируемости, отслеживающий все общие ресурсы до модулей, компонентов и приложений, использующих эти общие ресурсы, помогает подтвердить, что все потенциальные уязвимости были учтены [13]. Следует анализировать потенциальные нарушения обособления для устройств общей памяти, таких как постоянная память только для чтения (read-only memory, ROM), оперативная память (random-access memory, RAM), кэш, очереди и встроенные регистры микросхем. Аналогичным образом следует анализировать и влияние аппаратных отказов на общие и неразделяемые аппаратные компоненты. В разделе 3.5.2.5 DO-297 перечисляются некоторые распространенные потенциальные источники ошибок проектирования, которые могут влиять на обособление [3]:

a. Прерывания и запреты прерываний, программные и аппаратные.

- b. Циклы, например бесконечные циклы или косвенные незавершающиеся циклы вызовов.
 - c. Соответствие работе в реальном времени, например выход за пределы кадра по времени, воздействие на часы реального времени, повреждение счетчика или таймера, конвейеризация и кэширование, детерминированное планирование.
 - d. Поток управления, например ошибочное ветвление в обособленную или защищенную область, повреждение таблицы переходов, повреждение управления последовательностью процессора, повреждение адресов возврата, неисправимое повреждение аппаратного состояния, например `mask and halt`.
 - e. Конфликты по памяти, вводу и (или) выводу.
 - f. Совместное использование флагов данных.
 - g. Программные ловушки, например деление на ноль, невыполнимая инструкция, специальные команды программного прерывания, нераспознанная инструкция и завершение рекурсии.
 - h. Команды удержания (`hold-up commands`), то есть механизмы подстраховки производительности.
 - i. Потеря входных или выходных данных.
 - j. Повреждение входных или выходных данных.
 - k. Повреждение внутренних данных, например прямые или косвенные записи в память, переполнение таблицы, некорректная компоновка, вычисления с участием времени, поврежденная кэш-память.
 - l. Задержанные данные.
 - m. Наложения программ.
 - n. Последовательность обработки буферов.
 - o. Взаимодействие с внешними устройствами, например потеря данных, задержанные данные, некорректные данные, остановки по протоколу.
4. *Выявить потенциальные уязвимости, смягчаемые проектом.* Большинство уязвимостей смягчается проектом, особенно если надежное обособление проактивно учитывается на этапах разработки. Каждая уязвимость, смягчаемая проектом, должна трассироваться к требованию или требованиям, демонстрирующим это смягчение, то есть между уязвимостями и требованиями, обеспечивающими смягчение, должно существовать соответствие. Кроме того, каждое смягчение должно верифицироваться в ходе испыта-

ний.

5. *Выявить потенциальные уязвимости, смягчаемые процессом.* Некоторые потенциальные уязвимости могут смягчаться процессом, например использованием квалифицированного инструмента для проверки точности конфигурационных данных или стандарта проектирования, подтверждающего определенные назначения памяти.
6. *Выявить потенциальные уязвимости, которые не могут быть смягчены проектом или процессом.* Их потребуется довести до интегратора и, возможно, до разработчика приложения. Такая информация обычно документируется в техническом описании и в соответствующей информации для пользователя, чтобы интегратор или разработчик приложения могли предпринять необходимые действия.
7. *Координироваться со специалистами по безопасности и системам на протяжении всего анализа обособления.* Анализ обособления по сути является продолжением оценки безопасности и должен тесно координироваться со специалистами по системной безопасности.
8. *Обеспечить, чтобы все потенциальные уязвимости были смягчены, особенно те, которые не были устранены проектом или процессом.* Цель состоит в том, чтобы минимизировать действия по смягчению, которые должны выполняться интегратором или приложениями. Однако в некоторых случаях могут требоваться специальные действия. Например, интегратору может понадобиться выполнить специальные шаги верификации или наложить специальные ограничения через конфигурационные файлы.

Список литературы

1. Certification Authorities Software Team (CAST), *Guidelines for assessing software partitioning/protection schemes*, позиционный документ CAST-2 (февраль 2001).
2. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
3. RTCA DO-297, *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations* (Washington, DC: RTCA, Inc., November 2005).
4. J. Rushby, *Partitioning in avionics architectures: Requirements, mechanisms, and assurance*, DOT/FAA/AR-99/58 (Washington, DC: Office of Aviation Research, March 2000). Также опубликовано как NASA/CR-1999-209347 (Hampton, VA: Langley Research Center, March 2000).
5. B. L. Di Vito, *A formal model of partitioning for integrated modular avionics*, NASA/CR-1998-208703 (Hampton, VA: Langley Research Center, August 1998).

6. K. Driscoll, *Integrated modular avionics (IMA) requirements and development*, Integrated Modular Avionics Conference for the European Network of Excellence on Embedded Systems (Rome, Italy, 2007), p. 440.
7. S. H. VanderLeest, *ARINC 653 hypervisor*, IEEE Digital Avionics Systems Conference (Salt Lake City, UT, 2010), pp. 5.E.2-1-5.E.220.
8. J. Krodel and G. Romanski, *Real-time operating systems and component integration considerations in integrated modular avionics systems report*, DOT/FAA/AR-07/39 (Hampton, VA: Langley Research Center, August 2007).
9. V. Halwan and J. Krodel, *Study of commercial off-the-shelf (COTS) real-time operating systems (RTOS) in aviation applications*, DOT/FAA/AR-02/118 (Washington, DC: Office of Aviation Research, December 2002).
10. J. Littlefield-Lawwill and L. Kinnan, *System considerations for robust time and space partitioning in integrated modular avionics*, IEEE Digital Avionics Systems Conference (Orlando, FL, October 2008).
11. J. Krodel, *Commercial off-the-shelf real-time operating system and architectural considerations*, DOT/FAA/AR-03/77 (Washington, DC: Office of Aviation Research, February 2004).
12. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).
13. J. Krodel and G. Romanski, *Handbook for Real-Time Operating Systems Integration and Component Integration Considerations in Integrated Modular Avionics Systems*, DOT/FAA/AR-07/48 (Washington, DC: Office of Aviation Research, January 2008).

*CAST – это группа международных сертифицирующих органов, стремящаяся гармонизировать свои позиции по бортовому ПО и авиационной электронной аппаратуре в документах CAST.

*Тот же самый пример уже приводился ранее в этой главе, но повторяется здесь для полноты изложения.

Глава 22. Конфигурационные данные

Сокращения

Сокращение	Расшифровка
АТР	План приемочных испытаний
КК1	Категория контроля КК1
КК2	Категория контроля КК2
DAL	Уровень гарантии разработки
EASA	Европейское агентство по авиационной безопасности
FAA	Федеральное управление гражданской авиации США
IMA	Интегрированная модульная авионика
PR	Сообщение о проблеме
PSAC	План сертификации ПО
SAS	Итоговый отчет разработки ПО
SCI	Указатель конфигурации ПО
SCMP	План управления конфигурацией ПО
SQA	Гарантия качества ПО
SQAP	План гарантии качества ПО
SVP	План верификации ПО
XML	Расширяемый язык разметки

22.1 Введение

Многие системы, критичные по безопасности, проектируются как конфигурируемые. Конфигурационные данные дают гибкий, но при этом контролируемый способ конфигурирования и переконфигурирования системы. Данные, используемые для конфигурирования системы, в этой главе называются *конфигурационными данными*. В DO-178C (КТ-178С), а также в политике и руководящих материалах сертифицирующих органов для описания этих *конфигурационных данных* используются и другие термины, в том числе конфигурационные файлы, базы данных, бортовые системные базы данных, компоненты параметрических данных, элементы данных адаптации и т.д.

В этой главе объясняются некоторые термины, указания и рекомендации, связанные с конфигурационными данными. Аэронавигационные базы данных, например навигационные базы данных или базы данных рельефа, здесь не рассматриваются. Они обсуждаются в главе 23.

22.2 Терминология и примеры

В сертификационном меморандуме Европейского агентства по авиационной безопасности (EASA) CM-SWCEH-002 конфигурационные данные называются *configuration files* (конфигурационными файлами) и определяются так:

Файлы, содержащие параметры, используемые рабочей программой как вычислительные данные либо для активации или деактивации компонентов ПО (например, для адаптации программного обеспечения к одной из нескольких конфигураций воздушного судна или двигателя). Для конфигурационного файла иногда также используются термины “registry” (реестр) или “definition file” (файл определения). Конфигурационные файлы, такие как данные симвонологии, спецификации шин или файлы конфигурации воздушного судна или двигателя, отделяются от остального встроенного ПО в целях модульности и переносимости [1].

В разделе 15.2 приказа Федерального управления гражданской авиации США (FAA) 8110.49 (изменение 1) конфигурационные данные называются *airborne system databases* (бортовыми системными базами данных). В приказе поясняется, что бортовые системные базы данных: используются бортовой системой и утверждаются как часть типовой конструкции воздушного судна или двигателя. Эти базы данных могут влиять на пути выполнения исполняемого объектного кода, использоваться для активации или деактивации компонентов ПО и функций, адаптировать вычисления программного обеспечения к конфигурации воздушного судна либо использоваться как вычислительные данные. Бортовые системные базы данных могут состоять из файлов сценариев, интерпретируемых языков, структур данных или конфигурационных файлов, включая реестры (registry), программные опции, конфигурацию рабочей программы, модули конфигурации воздушного судна и программное обеспечение с выбором опций [2].

DO-178C развивает эти указания сертифицирующих органов, вводя понятия *parameter data item* (компонент параметрических данных) и *parameter data item file* (файл компонента параметрических данных), которые определяются так [3]:

- *Компонент параметрических данных* – набор данных, которые, будучи представлены в форме файла компонента параметрических данных, влияют на работу ПО без изменения исполняемого объектного кода и управляются как отдельная единица конфигурации. Примерами являются базы данных и таблицы конфигураций.
- *Файл компонента параметрических данных* – представление компонента параметрических данных, которое может напрямую использоваться процессором целевого вычислителя. Файл компонента параметрических данных представляет собой реализацию

компонента параметрических данных, содержащую определенные значения каждого элемента данных.

Конфигурационные данные отделены от исполняемого объектного кода. Иногда конфигурационные данные включаются в состав обозначения бортового ПО и поэтому входят в данные ЖЦ ПО по DO-178С, то есть перечисляются в Итоговом отчёте разработки ПО и Указателе конфигурации ПО бортового программного обеспечения. Другие конфигурационные данные могут иметь отдельное обозначение, отличное от обозначения бортового ПО, чтобы конфигурировать ПО под конкретные потребности воздушного судна. В такой ситуации конфигурационные данные могут иметь собственные данные ЖЦ ПО, включая отдельный Итоговый отчёт разработки ПО и отдельный Указатель конфигурации ПО, а также отдельное обозначение. Конфигурационные данные также могут загружаться в полевых условиях или модифицироваться пользователем.

Конфигурационные данные бывают самых разных форматов, включая текстовые файлы, двоичные данные, данные на расширяемом языке разметки (XML), таблицы поиска, базы данных и т.д. Ниже приведены примеры того, для чего могут использоваться конфигурационные данные:

- Определение символики для системы индикации.
- Задание параметров шины данных.
- Включение или отключение заранее запрограммированных функций, например программного обеспечения с выбором опций.
- Конфигурирование сети передачи данных.
- Программирование распределения ресурсов платформы интегрированной модульной авионики (Integrated Modular Avionics, IMA), например памяти, времени и общих ресурсов.
- Идентификация опций, специфичных для воздушного судна, например модуля индивидуализации.
- Калибровка датчиков или исполнительных механизмов.
- Конфигурирование операционной системы реального времени.

В разделе 2.5.1 DO-178С говорится следующее [3]:

Компоненты параметрических данных могут содержать данные, которые способны:

- а. Влиять на пути выполнения исполняемого объектного кода.
- б. Активировать или деактивировать компоненты ПО и функции.

- c. Адаптировать вычисления программного обеспечения к конфигурации системы.
- d. Использоваться как вычислительные данные.
- e. Устанавливать распределение времени и памяти между областями обособления.
- f. Предоставлять начальные значения компоненту ПО.

Конфигурационные данные обычно разрабатываются и контролируются одной из следующих сторон:

- *Системной или интеграционной командой производителя оборудования или авионики.* Конфигурация системы может определяться системной или интеграционной командой. Например, при определении требований к системным коммуникациям системная команда может определить сетевые соединения, а интеграционная команда может разрабатывать конфигурационные данные для реализации этих соединений.
- *Командой программного обеспечения производителя оборудования или авионики.* Команда ПО может включать конфигурационные данные в состав ПО изделия либо оформлять их как отдельную единицу конфигурации. Пример конфигурационных данных, входящих в состав программного пакета – таблица поиска для задания коэффициентов усиления. Пример конфигурационных данных, оформленных отдельно, – файл распределения ресурсов для приложения.
- *Производственным процессом производителя оборудования или авионики.* В некоторых ситуациях конфигурационные данные формируются как часть производственного процесса, а не процесса инженерной разработки и разработки программного обеспечения. Например, аппаратные конфигурационные данные, такие как серийный номер процессора и серийный номер изделия, могут вводиться как часть производственного процесса. Другой пример – калибровочные данные, вводимые до приемочных испытаний и отгрузки заказчику.
- *Процессом установки у производителя воздушного судна.* Иногда за часть конфигурационных данных отвечает производитель воздушного судна. Например, производитель воздушного судна может использовать настраиваемый модуль индивидуализации воздушного судна, чтобы конфигурировать самолет в соответствии с конкретными потребностями заказчика.
- *Процессом модификации у пользователя.* Некоторые конфигурационные данные могут быть модифицируемыми пользователем. Например, авиакомпания может использовать конфигурационные данные для определения того, какие параметры будут собираться в целях технического обслуживания.

22.3 Сводка указаний DO-178C по параметрическим данным

В DO-178B не было специальных указаний по конфигурационным данным. Однако, как уже отмечалось, в DO-178C для описания конфигурационных данных используются термины *parameter data item* и *parameter data item file*. В DO-248C поясняется, что компонент параметрических данных – это компонент ПО, содержащий только данные, без исполняемого объектного кода. Компонент параметрических данных также является единицей конфигурации. Бортовое программное обеспечение может состоять из одного или нескольких конфигурационных элементов исполняемого объектного кода и одного или нескольких конфигурационных компонентов параметрических данных. *Файл* компонента параметрических данных представляет собой реализацию компонента параметрических данных, содержащую определенные значения каждого элемента данных [4].

Компонентам параметрических данных назначается тот же уровень ПО, что и компоненту ПО, использующему эти данные. Файлы компонентов параметрических данных в DO-178C рассматриваются почти так же, как исполняемый объектный код: они определяются требованиями и должны быть верифицированы либо отдельно, либо как часть бортового ПО. В разделах 7 и 8 DO-178C поясняется, что файлы компонентов параметрических данных должны находиться под управлением конфигурацией ПО и оцениваться в ходе мероприятий по гарантии качества ПО, точно так же как и исполняемый объектный код. Аналогично, раздел 9 DO-178C относит файлы компонентов параметрических данных к данным типовой конструкции наряду с исполняемым объектным кодом.

Следующие цели DO-178C прямо ссылаются на файлы компонентов параметрических данных [3]:

- *Цель 7 таблицы A-3 DO-178C:* “Исполняемый объектный код и файлы компонентов параметрических данных, если таковые имеются, созданы и загружены в целевой вычислитель.” Эта цель была расширена по сравнению с DO-178B в DO-178C, чтобы охватить интеграцию параметрических данных в целевой вычислитель.
- *Цель 8 таблицы A-5 DO-178C:* “Файл компонента параметрических данных корректен и полон.” Эта цель появилась только в DO-178C. Она обеспечивает уверенность в том, что каждый элемент файла компонента параметрических данных удовлетворяет своим требованиям, имеет корректные значения и согласован с другими элементами данных.
- *Цель 9 таблицы A-5 DO-178C:* “Верификация файла компонента параметрических данных выполнена.” Она обеспечивает уверенность в том, что в ходе верификации были охвачены все элементы каждого файла компонента параметрических данных.

22.4 Рекомендации

Конфигурационные данные охватывают очень широкий спектр сценариев. У каждого сценария есть свои особенности, которые нужно учитывать в конкретном проекте. Тем не менее этот раздел содержит ряд общих рекомендаций, о которых полезно помнить. Многие из них основаны на DO-178C и на политике и руководящих материалах сертифицирующих органов.

Рекомендация 1: определить, как будут использоваться конфигурационные данные и какая политика и какие указания применимы. В зависимости от того, как используются конфигурационные данные, могут применяться указания по загружаемому в полевых условиях программному обеспечению, см. главу 18, по модифицируемому пользователем ПО, см. главу 19, либо по отключенному коду или ПО с выбором опций, см. главу 17. Следует учитывать указания DO-178C по компонентам параметрических данных, а также любые специальные указания сертифицирующего органа по конфигурационным данным. На момент написания книги сертификационные указания, относящиеся к конфигурационным данным, все еще развивались. Публикация DO-178C прояснила ожидаемый подход, однако из-за новизны указаний по компонентам параметрических данных возможны дополнительные разъяснения со стороны сертифицирующих органов. В целом конфигурационные данные рассматриваются как особый класс ПО; следовательно, к ним применима большая часть указаний, относящихся к бортовому ПО, хотя и с некоторыми поправками, о которых будет сказано далее.

Рекомендация 2: определить применимый уровень ПО или уровень гарантии разработки (Development Assurance Level, DAL). В ходе оценки безопасности рассматриваются возможные условия, которые возникнут, если конфигурационные данные окажутся ошибочными или поврежденными. Обычно конфигурационные данные рассматриваются в процессе оценки безопасности как программное обеспечение, поэтому им назначается уровень ПО. Однако если конфигурационные данные устанавливаются на системном уровне, может применяться уровень гарантии разработки. Обычно конфигурационным данным назначается тот же уровень ПО или уровень гарантии разработки, что и компоненту ПО или системы, который использует эти конфигурационные данные, либо наивысший из уровней, если в системе присутствует несколько уровней, как, например, в системе интегрированной модульной авионики.

В DO-178C поясняется, что уровень ПО компонента параметрических данных совпадает с уровнем ПО компонента ПО, который использует параметрические данные. Однако решающей остается оценка безопасности. Если применяется архитектурное снижение, возможна ситуация, когда уровень конфигурационных данных окажется ниже, но, скорее всего, это

будет довольно редкий случай.

Рекомендация 3: разработать подходящий процесс жизненного цикла для конфигурационных данных. Хотя конфигурационные данные похожи на программное обеспечение и к ним применима значительная часть указаний для бортового ПО, различия все же есть. Большинство проектов применяют цели DO-178С к конфигурационным данным. Однако для конфигурационных данных часто требуется только один уровень требований, например требования высокого уровня к ПО, а цели по структурному покрытию выполняются иначе, поскольку в конфигурационных данных нет условий или решений. В указаниях DO-178С, его целях и действиях для компонентов параметрических данных и файлов компонентов параметрических данных объясняется, как применять DO-178С к конфигурационным данным.

Рекомендация 4: разработать планы для документирования процесса. Планирование по конфигурационным данным может быть включено в состав планов по ПО либо оформлено отдельно. Обычно, если объем конфигурационных данных значителен или если конфигурационные данные будут регулярно изменяться отдельно от ПО, которое их использует, имеет смысл отделить конфигурационные данные и поддерживающие их данные ЖЦ ПО от данных исполняемого программного обеспечения. Когда конфигурационные данные оформляются отдельно от бортового ПО, пять планов, указанных в DO-178С, могут быть сжаты до одного или двух планов по конфигурационным данным, поскольку процессы в этом случае могут быть не такими сложными, как для бортового ПО, а команда может быть заметно меньше. Более чем в одном проекте мне встречались случаи, когда планы сводились к двум документам: (1) Плану сертификации ПО (Plan for Software Aspects of Certification, PSAC), который представляется сертифицирующему органу, и (2) другому плану, описывающему разработку, верификацию, управление конфигурацией и гарантию качества конфигурационных данных. Во многих случаях План управления конфигурацией ПО (Software Configuration Management Plan, SCMP) и План гарантии качества ПО (Software Quality Assurance Plan, SQAP) для бортового ПО могут также распространяться на конфигурационные данные, поэтому во втором плане на них просто даются ссылки. Независимо от того, как именно упакованы планы, раздел 4.2.j DO-178С поясняет, что при использовании *компонентов параметрических данных* в планах следует учитывать следующую дополнительную информацию [3]:

1. Способ использования компонентов параметрических данных.
2. Уровень ПО компонентов параметрических данных.
3. Процессы разработки, верификации и модификации компонентов параметрических данных, а также любую связанную с ними квалификацию инструментов.
4. Управление загрузкой ПО и совместимостью.

Рекомендация 5: определить требования к конфигурационным данным. Конфигурационные данные, как и любые другие данные, влияющие на функциональность воздушного судна, должны определяться требованиями. Часто для описания функциональности конфигурационных данных требуется только один уровень требований. DO-178C предлагает фиксировать требования к конфигурационным данным как требования высокого уровня к ПО (high-level software requirements); однако в ряде случаев для фиксации требований к конфигурационным данным использовались требования системного уровня или проектная спецификация конфигурационного файла. Способ упаковки здесь достаточно гибок; однако независимо от выбранной схемы следует помнить, что (1) требования, задающие конфигурационные данные, необходимы и (2) конфигурационные данные должны трассироваться на эти требования.

Рекомендация 6: трассировать конфигурационные данные на их требования. Как и в случае исходного кода, конфигурационные данные должны трассироваться на свои требования. Как будет сказано позже, данные трассируемости особенно важны для конфигурационных данных, поскольку анализ данных трассируемости иногда используется для подтверждения того, что отсутствуют непредусмотренные конфигурационные данные.

Рекомендация 7: разработать стандарты, обеспечивающие правильный формат конфигурационных данных. Чтобы конфигурационные данные могли быть прочитаны исполняемым объектным кодом, они должны иметь четко определенный формат. Обычно необходимы стандарты, задающие формат данных, допустимые значения, типы данных и т.д., чтобы направлять разработчика конфигурационных данных и обеспечивать согласованность и точность. В некоторых сценариях определенные константные значения могут требоваться всегда, например некоторые сетевые настройки для конкретного драйвера устройства. Такие ситуации могут регулироваться требованиями или стандартами разработки. Следует отметить, что стандарты нужно учитывать и при верификации конфигурационных данных, то есть проверять, что конфигурационные данные соответствуют стандартам разработки. При использовании инструментов именно инструменты часто обеспечивают соблюдение формата данных; однако необходимость в стандартах форматирования данных все равно может сохраняться.

Рекомендация 8: спроектировать исполняемое программное обеспечение как конфигурируемое. ПО, которое будет работать с конфигурационными данными, должно быть спроектировано так, чтобы использовать эти данные. Требования к ПО должны задавать структуру, атрибуты, допустимые диапазоны и т.д. для используемых конфигурационных данных. Во многих случаях исполняемое ПО выполняет ту или иную проверку корректности, чтобы убедиться, что конфигурационные данные находятся в заранее одобренных диапазонах и не были повреждены.

Рекомендация 9: верифицировать конфигурационные данные. Верификация конфигурационных данных включает несколько мероприятий, в том числе следующие:

1. *Убедиться, что требования к конфигурационным данным точны, полны, непротиворечивы, верифицируемы и корректны.* В CM-SWCEH-002 EASA это рассматривается как этап валидации (*validation*) [1]. Однако как бы это ни называлось, валидацией или верификацией, этот шаг должен быть выполнен. Обычно он достигается путем рассмотрения и анализа. Для сложных конфигурационных данных, например сетевой конфигурации воздушного судна, для подтверждения корректности и полноты данных часто используются моделирования. Для более простых конфигурационных данных может быть достаточно рассмотрения. Некоторые конфигурационные данные могут верифицироваться сочетанием рассмотрений и анализов.
2. *Проверить, что конфигурационные данные трассируются на соответствующие требования.* Важно убедиться, что все требования к конфигурационным данным реализованы и что все конфигурационные данные определяются требованиями.

Точная и полная трассируемость помогает это обеспечить. Важно не только убедиться, что трассируемость есть, но и убедиться, что это *правильная* трассируемость.

3. *Убедиться, что конфигурационные данные соответствуют требованиям.* К конфигурационным данным должны применяться тестирование на основе требований, а также рассмотрения и анализы, чтобы убедиться, что требования удовлетворены. Иногда конфигурационные данные верифицируются вместе с исполняемым объектным кодом. В других случаях они могут верифицироваться отдельно. В разделе 6.6 DO-178C поясняется, что если конфигурационные данные, в DO-178C они называются *файлами компонентов параметрических данных*, верифицируются отдельно от исполняемого объектного кода, который использует эти данные, то должно соблюдаться следующее [3]:

- Исполняемый объектный код разработан и верифицирован тестированием в нормальном диапазоне так, чтобы корректно обрабатывать все файлы компонентов параметрических данных, соответствующие их определенной структуре и атрибутам. Исполняемый объектный код обладает робастностью по отношению к структурам и атрибутам файлов компонентов параметрических данных.
- Все поведение исполняемого объектного кода, возникающее из-за содержимого файла компонента параметрических данных, может быть верифицировано. Структура данных ЖЦ ПО позволяет управлять компонентом параметрических данных отдельно.

4. *Убедиться, что в ходе верификации были охвачены все конфигурационные данные.* Что-

бы не осталось непредусмотренных конфигурационных данных, должно выполняться мероприятие, подтверждающее, что все конфигурационные данные верифицированы. Это похоже на анализ структурного покрытия, выполняемый для программного обеспечения. Однако поскольку критерии структурного покрытия обычно не применяются к базам данных, необходимо разработать другой подход. Он может быть похож на концепцию элементного анализа из DO-254 (KT-254), используемую для электронной аппаратуры, чтобы убедиться, что все *элементы* связаны с требованиями, которые были верифицированы. Для наших целей конфигурационные данные можно рассматривать как *элемент*; данные трассируемости могут использоваться для поддержки такого анализа и подтверждения того, что каждый элемент конфигурационных данных был протестирован. Независимо от выбранного подхода цель состоит в том, чтобы убедиться, что все конфигурационные данные должным образом верифицированы и что отсутствуют посторонние или мертвые данные.

5. *По возможности использовать квалифицированный инструмент верификации для проверки согласованности.* Поскольку верифицировать конфигурационные данные вручную бывает очень утомительно и не слишком эффективно, особенно если они представлены в растровом формате, инструменты могут быть весьма полезны. Например, квалифицированный инструмент может использоваться для подтверждения того, что система не посылает сообщения в порт передачи, что неразделяемые сегменты памяти не перекрываются, что сумма всех запросов ресурсов не превышает ресурсы, предоставленные системой, и так далее.
6. *Интегрировать конфигурационные данные с исполняемым объектным кодом в целевом вычислителе.* Интеграция может происходить в ходе тестирования, особенно если исполняемый объектный код и конфигурационные данные верифицируются совместно. Однако возможны ситуации, когда конфигурационные данные и исполняемый объектный код верифицируются отдельно. В этом случае потребуется дополнительный этап интеграции, чтобы убедиться, что исполняемый объектный код и конфигурационные данные совместно работают должным образом на конкретном целевом вычислителе.

Рекомендация 10: обеспечить совместимость исполняемого объектного кода и конфигурационных данных. Следует выполнить верификацию, подтверждающую совместимость конкретной конфигурации исполняемого объектного кода и конфигурационных данных. Должны быть предусмотрены мероприятия, позволяющие убедиться, что конфигурационные данные не повреждены и не являются неприемлемыми для исполняемого объектного кода, например не нарушают согласованный формат и диапазоны. Кроме того, совместимость должна быть задокументирована в соответствующем указателе конфигурации.

Рекомендация 11: применять управление конфигурацией к конфигурационным данным. Управление конфигурацией применяется к конфигурационным данным, а также к поддерживающим их данным ЖЦ ПО. Например, конфигурационные файлы должны иметь идентификацию конфигурации, изменения требований к конфигурационным данным и самих файлов конфигурации должны обрабатываться через запрос на изменение или сообщение о проблеме (Problem Report, PR), изменять конфигурационные данные должен только уполномоченный персонал и т.д. Кроме того, все аспекты категорий контроля КК1 или КК2 применяются к конфигурационным данным и поддерживающим их данным ЖЦ ПО точно так же, как и к бортовому программному обеспечению. См. главу 10, где рассматриваются ожидания DO-178C в части управления конфигурацией.

Рекомендация 12: если конфигурационные данные формируются в производственном процессе, обеспечить, чтобы этот процесс был контролируемым. По моему опыту, конфигурационные данные, вводимые в ходе производства, встречаются реже, чем конфигурационные данные, разрабатываемые инженерной командой в ходе разработки; однако для некоторых систем они могут требоваться. Производственный процесс должен быть четко определен и воспроизводим. Кроме того, процесс должен включать независимую верификацию ввода данных, особенно для более критичных данных, например уровней А и В. В некоторых сценариях План приемочных испытаний (Acceptance Test Plan, ATP) может трассироваться на требования к конфигурационным данным, чтобы обеспечить выявление любых ошибочных данных Планом приемочных испытаний. В других сценариях независимый человек может вручную верифицировать данные, например проверить серийные номера аппаратных компонентов и компонентов встроенного ПО, и подписать производственную запись.

Рекомендация 13: подготовить необходимую документацию, то есть данные ЖЦ ПО. Как и в случае бортового ПО, конфигурационные данные должны иметь требуемые данные ЖЦ ПО для поддержки их утверждения. Необходимы планы, данные разработки, данные верификации, данные управления конфигурацией и данные гарантии качества.

В общем случае для утверждения конфигурационных данных требуются те же данные ЖЦ ПО, которые определены в разделе 11 и приложении А DO-178C для демонстрации соответствия бортового ПО. Однако, как уже отмечалось, эти данные могут быть упакованы иначе, например планы могут быть объединены, может существовать только один уровень требований, а также применяются указания по компонентам параметрических данных из раздела 11 DO-178C.

Рекомендация 14: при модификации конфигурационных данных обновлять документацию и проводить повторную верификацию. При обновлении конфигурационных данных следу-

ет выполнять анализ влияния изменения, см. главу 10, и обновлять все затронутые данные. Кроме того, может потребоваться дополнительная верификация, чтобы убедиться, что обновленные конфигурационные данные по-прежнему совместимы с программным обеспечением, которое их использует.

Рекомендация 15: при использовании инструментов учитывать необходимость их квалификации. Инструменты часто используются для генерации и верификации конфигурационных данных, поскольку они могут устранять часть тривиальных ошибок, которые допускает человек. Если выход инструмента не верифицируется, инструмент, скорее всего, потребует квалификации. См. главу 13, где рассматривается квалификация инструментов.

Рекомендация 16: обеспечить документирование и утверждение инструкций по загрузке. Поскольку конфигурационные данные могут загружаться отдельно от программного обеспечения, важно, чтобы инструкции по загрузке и обновлению конфигурационных данных были хорошо документированы, воспроизводимы, недвусмысленны и утверждены. Инструкции по загрузке должны быть включены в Указатель конфигурации ПО (Software Configuration Index, SCI) либо на них должна быть дана ссылка из этого индекса.

Ссылки

1. European Aviation Safety Agency, *Software aspects of certification*, Certification Memorandum CM-SWCEH-002 (Issue 1, August 2011).
2. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Change 1, September 2011).
3. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
4. RTCA DO-248C, *Supporting Information for DO-178C and DO-278A* (Washington, DC: RTCA, Inc., December 2011).

Глава 23. Аэронавигационные данные

Сокращения

Сокращение	Расшифровка
AC	Консультативный циркуляр
AMDB	База данных картографической информации аэродрома
AMM	Подвижная карта аэродрома
AR	Требуется специальное разрешение
ASCII	Американский стандартный код для обмена информацией
DAL	Уровень гарантии разработки
DQR	Требования к качеству данных
EUROCAE	Европейская организация по оборудованию для гражданской авиации
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
FAQ	Часто задаваемые вопросы (frequently asked questions)
ICAO	Международная организация гражданской авиации
ISO	Международная организация по стандартизации
LOA	Письмо о принятии (letter of acceptance)
MASPS	Минимальные эксплуатационные стандарты авиационной системы
RNAV	Зональная навигация
RNP	Требуемые навигационные характеристики
TAWS	Система предупреждения о рельефе и опасном сближении с землей
TSO	Технический стандартный приказ
UML	Унифицированный язык моделирования
XML	Расширяемый язык разметки

23.1 Введение

Аэронавигационные данные – это “данные, используемые для таких аэронавигационных применений, как навигация, планирование полета, симуляция полета, предупреждение столкновения с землей и иные цели; они включают навигационные данные, а также данные о рельефе и препятствиях” [1]. *Аэронавигационная база данных* – это “совокуп-

ность данных, организованная и структурированная для удобства электронного хранения и извлечения в системе, поддерживающей бортовые или наземные аэронавигационные приложения” [1].

С аэронавигационными данными обращаются иначе, чем с конфигурационными данными, рассмотренными в главе 22. Конфигурационные данные из главы 22 утверждаются как часть данных типовой конструкции воздушного судна. Однако аэронавигационные данные часто не входят в данные типовой конструкции. Вместо этого загрузка таких данных нередко рассматривается как действие по техническому обслуживанию, указанное в инструкциях по поддержанию летной годности (Instructions for Continued Airworthiness, ICA). Параграф 43.3 раздела 14 Свода федеральных нормативных актов США (Code of Federal Regulations, CFR), обычно обозначаемого как 14 CFR, допускает такой подход и рассматривает подобные обновления как профилактическое техническое обслуживание (*preventative maintenance*). Есть как минимум две причины, по которым аэронавигационные данные рассматриваются иначе, чем конфигурационные.

Во-первых, аэронавигационные данные обычно требуют частых обновлений, которые непрактично выполнять в рамках процесса сертификации типа. Например, навигационные базы данных обновляются каждые 28 дней. Проходить процесс одобрения ПО по DO-178C (KT-178C) и дополнительную сертификацию типа или внесение изменений в сертификат типа каждые 28 дней практически невозможно. Базы данных рельефа обновляются реже, возможно три или четыре раза в год, но и это все равно слишком часто, чтобы процесс DO-178C был жизнеспособным.

Во-вторых, источником данных для таких баз часто выступает государственная организация, а не изготовитель авионики или воздушного судна. Приложение 15 Международной организации гражданской авиации (ICAO) устанавливает требования к государствам – участникам ICAO по всему миру, которые отвечают за сбор и передачу аэронавигационных данных через сборники аэронавигационной информации (Aeronautical Information Publications) [2]. “Каждое договаривающееся государство должно принять все необходимые меры, чтобы аэронавигационная информация и данные, которые оно предоставляет, были достаточными, имели требуемое качество (точность, разрешение и целостность) и предоставлялись своевременно для всей территории, за которую это государство несет ответственность” [1]. В прошлом требования ICAO в основном относились к навигационным данным. В последнее время в требования ICAO также включены требования к данным о рельефе. Как отмечалось в главе 22, DO-178C обычно не применяется к аэронавигационным данным. Однако RTCA DO-200A под названием *Standards for Processing Aeronautical Data* (стандарты обработки аэронавигационных данных) применяется именно к таким данным. Федеральное управление гражданской авиации США (FAA) признает DO-200A в консультативном циркуляре

(Advisory Circular, AC) 20-153A, *Acceptance of Aeronautical Data Processes and Associated Databases* (принятие процессов обработки аэронавигационных данных и связанных баз данных), а также в различных других документах, включая технические стандартные приказы (Technical Standard Order, TSO) TSO-146c и TSO-151b, AC 20-138B, AC 90-101A, Order 8110.55A и Order 8110.49.

В этой главе рассматриваются DO-200A, AC 20-153A и ряд других документов Федерального управления гражданской авиации США и отрасли, относящихся к аэронавигационным данным, чтобы дать общее представление об ожиданиях в этой области.

Примечание переводчика. Для практической ориентации в российских документах полезно учитывать соответствие: DO-200A – КТ-200А.

23.2 DO-200A: Standards for Processing Aeronautical Data

Документ RTCA DO-200A, а также его эквивалент ED-76 Европейской организации по оборудованию для гражданской авиации (EUROCAE), признается *основным стандартом* обеспечения качества аэронавигационных данных. На DO-200A ссылаются, среди прочего, следующие документы FAA:

- AC 20-153A – *Acceptance of Aeronautical Data Processes and Associated Databases*.
- Order 8110.55A – *How to Evaluate and Accept Processes for Aeronautical Database Suppliers*.
- Order 8110.49 – *Software Approval Guidelines*.
- AC 20-138B – *Airworthiness Approval of Positioning and Navigation Systems*.
- AC 90-101A – *Approval Guidance for Required Navigation Performance (RNP) Procedures with Authorization Required (AR)*.
- TSO-C151b – *Terrain Awareness and Warning System (TAWS)*.
- TSO-C146c – *Stand-Alone Airborne Navigation Equipment Using the Global Positioning System Augmented By The Satellite Based Augmentation System*.

В этом разделе приведен обзор DO-200A. В следующем разделе рассматриваются AC 20-153A и Order 8110.55A Федерального управления гражданской авиации США, которые определяют процесс получения письма о принятии (letter of acceptance, LOA) для процесса, соответствующего DO-200A.

DO-200A рассматривается как минимальный стандарт и руководящий материал по гарантии качества обработки и управлению качеством аэронавигационных данных, используемых для навигации, планирования полета, предупреждения столкновения с землей, симуляции

полета и т.д. Результатом процесса, соответствующего DO-200A, является база данных, распространяемая пользователю для загрузки в его оборудование.

DO-200A определяет требования и рекомендации, обеспечивающие надлежащий уровень гарантии для аэронавигационных данных. В DO-200A уровень гарантии (*assurance level*) определяется как “степень уверенности в том, что элемент данных не был поврежден при хранении или передаче. Его можно отнести к одному из трех уровней: 1, 2 и 3, где 1 – наивысшая степень уверенности” [1]. Как и в случае программных уровней DO-178C, уровни гарантии DO-200A определяются потенциальным влиянием поврежденных данных на безопасность. DO-201A, рассматриваемый в разделе 23.5.1, определяет уровни гарантии для аэронавигационной информации, связанной с зональной навигацией (area navigation, RNAV). Уровни гарантии для применений данных, не охваченных DO-201A, должны определяться конечным пользователем или поставщиком приложения на основе процесса оценки безопасности [2].

В таблице 23.1 показаны три уровня гарантии DO-200A и их связь с уровнями критичности ICAO и категориями отказных состояний. Также показаны соответствующие уровни гарантии разработки (DAL) или уровни ПО. Классификации ICAO: critical, essential и routine, определяются следующим образом [3]:

- *Критические данные (уровень гарантии 1)* – данные, ошибочность которых не позволила бы продолжать безопасный полет и посадку либо снизила бы способность справляться с неблагоприятными условиями эксплуатации до такой степени, что произошло бы значительное уменьшение запасов безопасности или функциональных возможностей. При попытке использовать поврежденные критические данные существует высокая вероятность того, что воздушное судно окажется в угрожающей жизни ситуации.

Таблица 23.1 Уровни гарантии DO-200A

Уровень гарантии DO-200A Связанное требование к данным, предоставляемым государством (ICAO)	Категория отказного состояния DAL или программный уровень
1 Critical	Катастрофический или опасный /серьезный-значительный A, B
2 Essential	Значительный или незначительный C, D
3 Routine	Нет влияния на безопасность E

- *Существенные данные (уровень гарантии 2)* – данные, ошибочность которых снижает способность справляться с неблагоприятными условиями эксплуатации до такой степени, что происходит заметное уменьшение запасов безопасности. При попытке использовать поврежденные существенные данные существует низкая вероятность того,

что воздушное судно окажется в угрожающей жизни ситуации.

- Обычные данные (уровень гарантии 3) – данные, ошибочность которых не приведет к значимому снижению безопасности самолета. При попытке использовать поврежденные обычные данные существует очень низкая вероятность того, что воздушное судно окажется в угрожающей жизни ситуации.

В DO-200A используется понятие цепочки аэронавигационных данных для описания пути, который проходят эти данные: “Цепочка аэронавигационных данных – это серия взаимосвязанных звеньев, каждое из которых выполняет функцию, обеспечивающую создание, передачу и использование аэронавигационных данных для конкретной цели” [1]. Обычно в нее входят пять основных звеньев: формирование (origination), передача (transmission), подготовка (preparation), интеграция в приложение (application integration) и конечное использование (end use) [1]. В одном звене могут участвовать несколько организаций, например в фазах подготовки и передачи могут существовать *sublinks* (подзвенья), а одна организация может выполнять сразу несколько типов звеньев, например одна компания может формировать, подготавливать и передавать данные. Первичный источник (originator) – это “первая организация в цепочке аэронавигационных данных, которая принимает на себя ответственность за данные. Например, государство или организация, соответствующая RTCA DO-200A / EUROCAE ED-76” [1]. Конечный пользователь (end user) – это “последний пользователь в цепочке аэронавигационных данных. Конечными пользователями аэронавигационных данных обычно являются эксплуатанты воздушных судов, подразделения авиакомпаний по планированию, поставщики обслуживания воздушного движения, поставщики услуг симуляции полета, самолетостроители, системные интеграторы и регулирующие органы” [1]. На рисунке 23.1 показана типичная цепочка аэронавигационных данных.

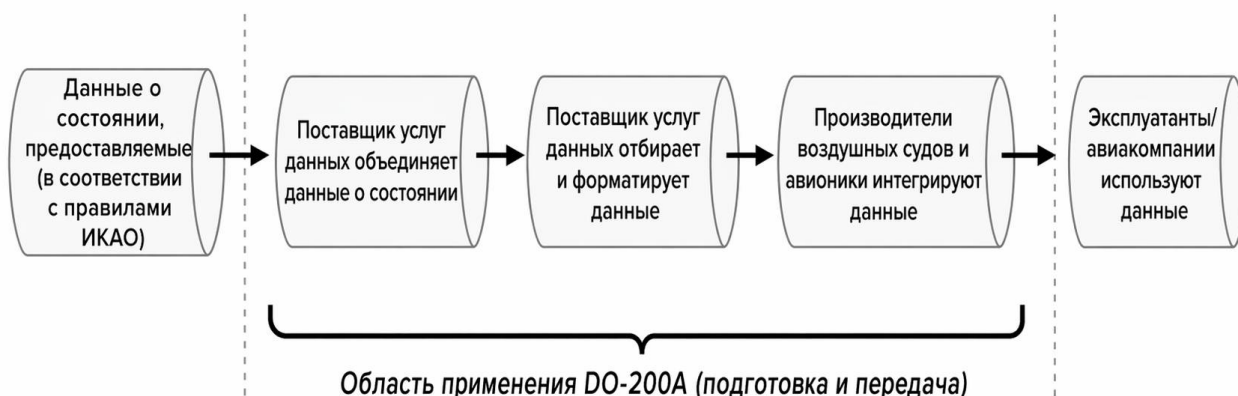


Рисунок 23.1 Типичная цепочка аэронавигационных данных.

На каждом звене цепочки аэронавигационных данных данные должны удовлетворять следующим семи характеристикам качества, определяемым предполагаемой функцией, которая будет использовать эти данные: (1) точность (accuracy), (2) разрешение (resolution), (3) уровень гарантии (assurance level, определяемый по результатам оценки безопасности), (4) трассируемость (traceability, до источника данных), (5) своевременность (timeliness, чтобы поддерживать актуальность и действительность данных), (6) полнота (completeness, то есть предоставление всех необходимых данных) и (7) формат (format, соответствующий назначению данных, включая разрешение при передаче). Эти требования к качеству данных определяются на основе функции, которую должны поддерживать данные.

DO-200A сосредоточен на звеньях подготовки (*preparation*) и передачи (*transmission*) в цепочке аэронавигационных данных. Хотя звенья формирования (*origination*), интеграции в приложение (*application integration*) и конечного использования (*end use*) упоминаются для контекста, они считаются выходящими за рамки DO-200A.

Функциональное звено подготовки аэронавигационных данных включает четыре фазы:

1. *Assemble* – сбор данных от поставщиков.
2. *Translate* – изменение формата представления информации, часто совмещаемое с фазой сборки.
3. *Select* – выбор нужных данных из собранного массива аэронавигационных данных.
4. *Format* – преобразование данных в формат, приемлемый для следующего функционального звена.

Функциональное звено передачи аэронавигационных данных включает две фазы:

- *Receive* – прием, верификация и валидация данных.
- *Distribute* – последняя фаза модели обработки, входящая в состав звена передачи.

На каждой фазе обработки данные верифицируются, а любые проблемы документируются в отчете об ошибке. При необходимости выполняются корректирующие действия. Если в данных из доверенного источника (*trusted source*), например государства – члена ИКАО, обнаружена ошибка, об этом необходимо сообщить доверенному источнику. Однако часто бывает трудно добиться, чтобы доверенный источник немедленно исправил данные. Нередко после подтверждения ошибки исправление выполняет сама организация, применяющая процесс, соответствующий DO-200A.

Раздел 2 DO-200A определяет указания для каждой организации, чтобы обеспечить соответствие данных семи характеристикам качества, перечисленным выше. Каждая организация

в цепочке данных отвечает за следующее:

- разработку плана соответствия;
- определение требований к качеству данных;
- определение процедур обработки данных;
- обеспечение того, чтобы изменение данных не выполнялось без должной координации с первичным источником;
- поддержание управления конфигурацией;
- обеспечение необходимых навыков и компетенций персонала;
- выполнение оценки и, при необходимости, квалификации инструментов;
- разработку и внедрение процедур управления качеством.

В зависимости от уровня гарантии объем требуемых работ по валидации и верификации различается. В DO-200A валидация и верификация определяются следующим образом [1]:

- *Validation* – мероприятие, в ходе которого проверяется, что элемент данных имеет значение, полностью применимое к идентичности, присвоенной этому элементу данных, либо что набор элементов данных приемлем для своей цели.
- *Verification* – мероприятие, в ходе которого текущее значение элемента данных сравнивается со значением, предоставленным изначально.

В общем случае для уровня гарантии 1 требуются и валидация путем применения (validation by application, то есть путем реального использования проверяемых данных), и верификация; для уровня гарантии 2 валидация путем применения не требуется, но требуется верификация; а для уровня гарантии 3 валидация и верификация рекомендуются, но не являются обязательными [1].

23.3 Консультативный циркуляр FAA 20-153A

Как в предыдущем разделе был дан обзор DO-200A высокого уровня, так и этот раздел содержит краткое изложение консультативного циркуляра FAA AC 20-153A.

AC 20-153, предшественник AC 20-153A, распространялся только на навигационные базы данных. В 2010 году FAA расширило действие этого циркуляра на другие типы аэронавигационных данных, включая базы данных рельефа, препятствий и карт аэродромов. Каждый из типов баз данных, охватываемых AC 20-153A, кратко поясняется ниже. Циркуляр может применяться и к другим базам данных, однако такие случаи следует тесно координировать с сертифицирующим органом.

- *Навигационная база данных* – “любые навигационные данные, хранящиеся в электронном виде в системе, поддерживающей навигационные приложения. Навигационные данные – это информация, предназначенная для помощи пилоту в определении положения воздушного судна относительно планов полета, наземных ориентиров и навигационных точек ... а также объектов на поверхности аэродрома” [2].
- *База данных рельефа* – “любые данные, хранящиеся в электронном виде в системе, поддерживающей приложения, связанные с рельефом. Данные рельефа включают естественную поверхность земли, исключая созданные человеком препятствия” [2].
- *База данных препятствий* – “любые данные, хранящиеся в электронном виде в системе, поддерживающей приложения, связанные с препятствиями. Данные о препятствиях включают любой естественный или искусственный неподвижный объект, имеющий вертикальную значимость относительно соседних и окружающих объектов и рассматриваемый как потенциальная опасность для безопасного полета воздушного судна” [2].
- *База данных картографической информации аэродрома* – “любые навигационные данные, хранящиеся в электронном виде в системе, поддерживающей приложения карт аэродрома. Данные картографической информации аэродрома – это информация, предназначенная для помощи пилоту в определении положения воздушного судна относительно объектов на поверхности аэродрома” [2].

АС 20-153А содержит указания для поставщиков аэронавигационных услуг, изготовителей оборудования или авионики и (или) эксплуатантов, необходимые для получения письма о принятии (letter of acceptance, LOA). Письмо о принятии выдается Федеральным управлением гражданской авиации США и подтверждает соответствие АС 20-153А и DO-200А в части обработки аэронавигационных данных. “Письмо о принятии официально документирует, что базы данных поставщика создаются в соответствии с RTCA/DO-200А или, для некоторых ранее сложившихся систем, в соответствии с RTCA/DO-200” [2]. В АС 20-153А выделяются два типа писем о принятии:

- *Письмо о принятии типа 1*. Письма о принятии типа 1 предназначены прежде всего для поставщиков данных, являющихся поставщиками услуг данных. “Письмо о принятии типа 1 признает соответствие поставщика данных RTCA/DO-200А без установленной совместимости с системой воздушного судна... Такое письмо о принятии может выдаваться поставщикам данных, эксплуатантам, изготовителям авионики и другим лицам” [2].
- *Письмо о принятии типа 2*. “Письма о принятии типа 2 основаны на требованиях, обеспечивающих совместимость с конкретными системами или оборудованием, и предназначены для поставщиков данных, которые являются изготовителями авионики или ин-

теграторами приложений... Для поставщиков данных типа 2 действуют дополнительные требования, обеспечивающие совместимость поставляемой базы данных с требованиями к качеству данных (DQR), необходимыми для поддержки предполагаемой функции, одобренной для целевого приложения” [2].[*]

АС 20-153А объясняет процесс подачи заявки на письмо о принятии и применение DO-200А, а также дополнительные требования для заявок на письма о принятии типов 1 и 2. Циркуляр также содержит дополнительные замечания по процессам DO-200А. В нем даны указания для эксплуатантов, изготовителей оборудования и авионики, то есть поставщиков приложений, а также для поставщиков услуг данных. В консультативном циркуляре также поясняется, что договаривающимся государствам рекомендуется следовать DO-200А, а также иным применимым указаниям по качеству данных, например DO-201А, DO-272В и DO-276А.

Раздел 14 АС 20-153А поясняет, что данным, полученным из источника, соответствующего DO-200А, либо от договаривающегося государства, можно доверять. Однако данные от иных поставщиков должны быть верифицированы и валидированы в соответствии с требуемым уровнем гарантии. Согласно циркуляру, DO-272В, раздел 3.9, содержит приемлемые методы верификации и валидации данных карт аэродромов [4].[*] Аналогично, согласно циркуляру, DO-276А, разделы 6.1.4, 6.1.5, содержит приемлемые методы верификации и валидации данных о рельефе и препятствиях [5].

Раздел 17 АС 20-153А вносит исправление в значения вероятностей необнаружения ошибок, указанные в DO-200А. В циркуляре отмечается, что для уровня гарантии 1 вероятность необнаруженного повреждения должна быть меньше либо равна 10^{-9} , а не 10^{-8} , как указано в DO-200А. Аналогично, для уровня гарантии 2 вероятность необнаруженного повреждения должна быть меньше либо равна 10^{-5} , а не 10^{-4} , как указано в DO-200А. Эти вероятности лучше согласуются с требованиями XX.1309 на уровне воздушного судна [2].[†]

Приложение 3 к АС 20-153А содержит матрицу соответствия, которую поставщики аэронавигационных баз данных могут использовать, чтобы убедиться в выполнении требований DO-200А и АС 20-153А до подачи заявки на письмо о принятии.

Помимо АС 20-153А Федеральное управление гражданской авиации США опубликовало приказ 8110.55А, *How to Evaluate and Accept Processes for Aeronautical Database Suppliers* [6]. Этот приказ содержит указания для подразделений Aircraft Certification Office Федерального управления гражданской авиации США по оценке и принятию аэронавигационных процессов и выдаче писем о принятии. Хотя приказ предназначен главным образом для сотрудников Федерального управления гражданской авиации США, он дает поставщи-

кам аэронавигационных баз данных некоторое представление о том, чего ожидать во время аудитов этого органа и как к ним готовиться.

23.4 Инструменты, используемые для обработки аэронавигационных данных

Из-за большого объема данных и необходимости воспроизводимости обработка аэронавигационных данных в значительной степени опирается на инструменты. Обычно для обеспечения целостности аэронавигационных данных по мере их прохождения по цепочке используются сочетание проверок целостности, например проверок с использованием циклического избыточного кода (CRC), и квалифицированные инструменты. Иногда инструменты запускаются параллельно, а результаты сравниваются, чтобы избежать необходимости квалификации. Все инструменты должны быть идентифицированы и оценены. Инструменты, выход которых не верифицируется, могут требовать квалификации. DO-200A и AC 20-153A ссылаются на критерии квалификации инструментов из раздела 12.2 DO-178B. Допустимо также использовать раздел 12.2 DO-178C. Как пояснялось в главе 13, раздел 12.2 DO-178C привлекает DO-330, *Software Tool Qualification Considerations* (вопросы квалификации программных инструментов). DO-330 писался так, чтобы быть применимым к нескольким предметным областям, в том числе к области аэронавигационных данных. Пункт D.7 раздела часто задаваемых вопросов (FAQ) документа DO-330 в некоторых сценариях может содержать полезную информацию об инструментах обработки аэронавигационных данных. В этом разделе FAQ описывается, как выход неквалифицированного инструмента может верифицироваться квалифицированным инструментом, чтобы обеспечить корректность выхода неквалифицированного инструмента.

23.5 Другие отраслевые документы, относящиеся к аэронавигационным данным

В DO-200A и AC 20-153A приводятся ссылки на несколько других отраслевых документов. Чтобы завершить обзор руководящих материалов по аэронавигационным базам данных, ниже дано краткое изложение этих документов. Сначала представлены документы RTCA в последовательном порядке, затем два документа ARINC.

23.5.1 DO-201A: Standards for Aeronautical Information

DO-201A объединяет общие и специальные требования к аэронавигационным данным с акцентом на операции зональной навигации (RNAV) в воздушном пространстве с требуемыми навигационными характеристиками (RNP). В нем приводятся общие требования и стандарты для аэронавигационных данных, включая точность, разрешение, правила вычислений, соглашения по наименованиям и своевременное распространение готовых данных [3]. Также приводятся специальные эксплуатационные требования и стандарты; их следует

учитывать при разработке процедур для маршрута, прибытия, вылета, захода на посадку и аэродромной среды. Как отмечалось ранее, DO-201A определяет уровни гарантии для аэронавигационной информации, относящейся к RNP. Уровни гарантии для применений данных, не охваченных DO-201A, должны определяться конечным пользователем или поставщиком приложения на основе процесса оценки безопасности. Эквивалентом DO-201A в EUROCAE является ED-77.

23.5.2 DO-236B: Minimum Aviation System Performance Standards: Required Navigation Performance for Area Navigation

DO-236B содержит минимальные эксплуатационные стандарты авиационной системы (minimum aviation system performance standards, MASPS) для систем зональной навигации (RNAV), работающих в среде с требуемыми навигационными характеристиками (RNP). Эти стандарты предназначены для разработчиков, изготовителей и установщиков авионики, а также поставщиков услуг и пользователей таких систем для эксплуатации по всему миру [7]. DO-236B ссылается на соответствие DO-201A и DO-200A для навигационных баз данных, используемых в системах RNAV. Кроме того, DO-201A содержит указания по требованиям, поддерживающим RNAV и операции RNP, описанные в DO-236B. Эквивалентом DO-236B в EUROCAE является ED-75B.

23.5.3 DO-272C: User Requirements for Aerodrome Mapping Information

DO-272C задает минимальные требования, применимые к содержанию, формированию, публикации и обновлению картографической информации аэродромов. Он также содержит указания по оценке соответствия этим требованиям и по определению необходимого уровня доверия. В нем определяется минимальный набор требований к качеству данных, который может использоваться для отображения карт аэропортов [2,8]. Эквивалентом DO-272C в EUROCAE является ED-99C.

23.5.4 DO-276A: User Requirements for Terrain and Obstacle Data

DO-276A определяет минимальные пользовательские требования, применимые к данным о рельефе и препятствиях. Он включает минимальный перечень атрибутов, связанных с данными о рельефе и препятствиях, и описывает соответствующие ошибки, которые может потребоваться учитывать [5]. Эквивалентом DO-276A в EUROCAE является ED-98A.

23.5.5 DO-291B: Interchange Standards for Terrain, Obstacle, and Aerodrome Mapping Data

DO-291B[*] используется совместно с DO-272C и DO-276A, которые задают пользовательские требования к содержанию и качеству баз данных рельефа, препятствий и аэродромной картографической информации. DO-291B задает требования к формату обмена данными, сформированными в соответствии с DO-272C и DO-276A. DO-291B был основан на серии

стандартов ISO 19100 по географической информации (geographic information) применительно к базам данных рельефа, препятствий и картографической информации аэродромов, используемым в авиации. Стандарт задает требования к области применения, идентификации, метаданным, содержанию, системе координат, качеству данных, сбору данных и сведениям по сопровождению [9]. Эквивалентом DO-291B в EUROCAE является ED-119B. DO-291B также связан со стандартом ARINC 816, который рассматривается в разделе 23.5.7.

23.5.6 ARINC 424: Standard, Navigation System Database

“Стандарт ARINC 424 является отраслевым стандартом обмена навигационными данными между поставщиками данных и поставщиками авионики уже более 30 лет” [10].

Он служит рекомендуемым стандартом отрасли воздушных перевозок для подготовки файлов эталонных данных бортовых навигационных систем.

Данные в этих файлах предназначены для объединения с эксплуатационным ПО бортового навигационного вычислителя с целью формирования носителей, используемых такими вычислителями на борту воздушных судов... Это позволяет поставщикам баз данных, системам авионики и другим пользователям баз данных выполнять полеты и планировать процедуры полета в соответствии с тем, как они заданы разработчиками процедур [11].

Первоначальная версия стандарта ARINC 424 была опубликована в 1975 году. На момент написания книги последней версией ARINC 424 является 424-19. Однако комитет активно работает над ARINC 424A. Текущий формат ARINC 424 определяет текстовые файлы фиксированной длины, по 132 символа в строке. Эти файлы преобразуются в двоичные образы и загружаются в конкретную систему управления полетом. Из-за зависимостей от целевого вычислителя каждый тип вычислителя управления полетом имеет свой двоичный образ. Это приводит к ситуации, когда у одной авиакомпания, эксплуатирующей несколько типов воздушных судов и систем управления полетом, могут быть десятки пакетов баз данных, хотя исходные данные для их создания идентичны. Цель комитета по ARINC 424A состоит в разработке открытого стандарта базы данных, напрямую читаемого вычислителем управления полетом, без необходимости в двоичном образе, специфичном для вычислителя управления полетом. Это позволит авиакомпании получать данные из единого источника. В ARINC 424A предлагается использовать модель унифицированного языка моделирования (Unified Modeling Language, UML), которая будет выдавать тот же формат ASCII (американский стандартный код для обмена информацией), что и традиционный стандарт ARINC 424. Однако эта модель также будет использоваться для автоматической генерации формата расширяемого языка разметки (eXtensible Markup Language, XML), который можно преобразовать в двоичный формат XML (binary XML) для уменьшения размера файлов и упрощения разбора. Комитет стремится сохранить текущий формат ARINC 424 и одновременно обес-

печить более стандартизованные результирующие файлы, которые можно использовать в нескольких системах управления полетом [10].

23.5.7 ARINC 816-1: Embedded Interchange Format for Airport Mapping Database

Стандарт ARINC 816-1 определяет формат баз данных, используемых для подвижных карт аэродрома (airport moving maps, AMM). Такие карты потенциально повышают безопасность, уменьшая число выездов на ВПП и рулежные дорожки, вызванных потерей ситуационной осведомленности экипажа. Стандарт ARINC 816-1 упрощает обращение с данными и предоставляет возможности, полезные для отображения подвижных карт аэродрома [12].

ARINC 816-1 связан с DO-272C и DO-291B, которые были описаны ранее. На рисунке 23.2 показана взаимосвязь этих трех документов. DO-272C определяет требования к содержанию, качеству и обработке баз данных картографической информации аэродрома (AMDB). DO-291B определяет требования к формату обмена данными для этих баз, обеспечивая обмен такими базами между первичными источниками данных и интеграторами. ARINC 816-1 задает стандарт базы данных для встроенных систем авионики. Стандартизованный формат позволяет конечным пользователям выбирать между различными интеграторами баз данных независимо от поставщика авионики. Это позволяет интеграторам баз данных преобразовывать данные аэродрома непосредственно в формат, соответствующий спецификации конечной системы [13].

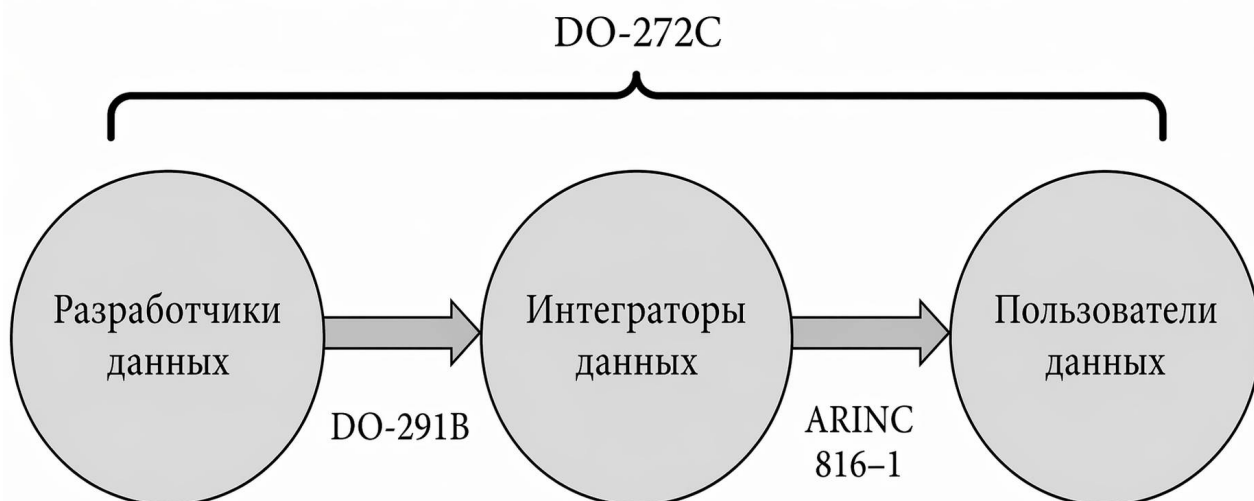


Рисунок 23.2 Взаимосвязь между DO-272C, DO-291B и ARINC 816-1.

Ссылки

1. RTCA/DO-200A, *Standards for Processing Aeronautical Data* (Washington, DC: RTCA, Inc., September 1998).
2. Federal Aviation Administration, *Acceptance of Aeronautical Data Processes and Associated Databases*, Advisory Circular 20-153A (September 2010).

3. RTCA/DO-201A, *Standards for Aeronautical Information* (Washington, DC: RTCA, Inc., April 2000).
4. RTCA DO-272B, *User Requirements for Aerodrome Mapping Information* (Washington, DC: RTCA, Inc., April 2009).
5. RTCA DO-276A, *User Requirements for Terrain and Obstacle Data* (Washington, DC: RTCA, Inc., August 2005).
6. Federal Aviation Administration, *How to Evaluate and Accept Processes for Aeronautical Database Suppliers*, Order 8110.55A (November 2011).
7. RTCA DO-236B, *Minimum Aviation System Performance Standards: Required Navigation Performance for Area Navigation* (Washington, DC: RTCA, Inc., October 2003).
8. RTCA DO-272C, *User Requirements for Aerodrome Mapping Information* (Washington, DC: RTCA, Inc., September 2011).
9. RTCA DO-291B, *Interchange Standards for Terrain, Obstacle, and Aerodrome Mapping Data* (Washington, DC: RTCA, Inc., September 2011).
10. C. Pschierer, J. Kasten, J. Schiefele, H. Lepori, P. Bruneaux, A. Bourdais, and R. Andreae, *ARINC 424A-A next generation navigation database specification*, IEEE Digital Avionics Systems Conference (Orlando, FL, 2009), 4.D.2-1-4.D.2.10.
11. Aeronautical Radio, Inc., *Navigation system database*, ARINC Specification 424-19 (Annapolis, MD: Airlines Electronic Engineering Committee, December 2008).
12. C. Pschierer and J. Schiefele, *Open standards for airport databases- ARINC 816*, IEEE Digital Avionics Systems Conference (Dallas, TX, 2007), 2.B.6-1-2.B.6-8.
13. Aeronautical Radio, Inc., *Embedded interchange format for airport mapping database*, ARINC Specification 816-1 (Annapolis, MD: Airlines Electronic Engineering Committee, November 2007).

*Скобки добавлены для пояснения.

*Следует отметить, что после публикации AC 20-153A документ DO-272B был заменен на DO-272C.

† XX может означать части 23, 25, 27 или 29 раздела 14 Свода федеральных нормативных актов США (Code of Federal Regulations, CFR), обычно сокращаемого как 14 CFR.

*AC 20-153A ссылается только на DO-291A, поскольку этот консультативный циркуляр был опубликован до выхода DO-291B.

Глава 24. Повторное использование программного обеспечения

Сокращения

Сокращение	Расшифровка
AC	Консультативный циркуляр
CAST	Группа сертифицирующих органов по ПО
COTS	Готовое коммерческое ПО
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
PDS	Ранее разработанное ПО
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
RSC	Повторно используемый компонент ПО
RTOS	ОСРВ (операционная система реального времени, real-time operating system)
SOUP	ПО неизвестного происхождения
U.S.	Соединенные Штаты

24.1 Введение

Повторное использование программного обеспечения является важной темой, поскольку большинство программных проектов, по крайней мере в авиационной отрасли, представляют собой производные уже существующих систем. Новые, создаваемые с нуля авионика, электрические системы, планеры и двигатели на деле встречаются довольно редко. DO-178C (KT-178C) поощряет хорошо организованный и дисциплинированный процесс, который поддерживает не только первоначальную сертификацию и соответствие нормативным требованиям, но и последующие модификации и сопровождение.

Обсуждать повторное использование программного обеспечения важно еще и потому, что существуют довольно широко известные последствия неудачного повторного использования. Один из примеров – авария ракеты-носителя Ariane 5. Программное обеспечение, использованное в Ariane 5, изначально предназначалось для Ariane 4 и работало на этой платформе корректно. Однако характеристики запуска ракет Ariane 4 и Ariane 5 различались.

Неправильное повторное использование программного обеспечения Ariane 4 привело к разрушению Ariane 5 [1]. В своей книге *Safeware* Нэнси Левесон объясняет, что представление о том, будто повторное использование программного обеспечения повышает безопасность, является мифом. Она приводит ряд связанных с безопасностью проблем, возникших из-за

такого повторного использования; три из них приведены ниже. Во-первых, медицинское устройство Therac-25 повторно использовало части своего предшественника, Therac-20. Ошибка существовала в программном обеспечении Therac-20, но не имела для его работы серьезных последствий, кроме разве что периодически перегоравшего предохранителя. К сожалению, при использовании в Therac-25 эта ошибка привела к массивным передозировкам излучения и по меньшей мере к двум смертям. Повторное использование программного обеспечения не было единственной причиной проблемы Therac-25, но стало существенным сопутствующим фактором. Во-вторых, программа управления воздушным движением, использовавшаяся в Соединенных Штатах, была повторно использована в Великобритании. В этом случае британские пользователи не учли различия по долготе между США и Великобританией. В-третьих, проблемы возникли, когда программа, написанная для воздушного судна в северном полушарии и выше уровня моря, была повторно использована в южном полушарии или ниже уровня моря [2].

Примечание переводчика. Ariane 5 – европейская ракета-носитель; ее первый пуск 4 июня 1996 года закончился аварией примерно через 40 секунд после старта из-за отказа инерциальной системы, где повторно использованный код Ariane 4 оказался несовместим с новым профилем полета. Therac-25 – медицинский линейный ускоритель для лучевой терапии; в 1985–1987 годах несколько пациентов получили опасные передозировки излучения, а среди причин расследователи выделяли ошибки программного управления, недостаточные аппаратные блокировки безопасности и повторное использование кода от предыдущих моделей. Оба примера здесь важны как напоминание: повторно используемое программное обеспечение остается безопасным только в том контексте, для которого проверены его допущения. Программное обеспечение необходимо повторно использовать с крайней осторожностью. В этой главе тема рассматривается через два специальных понятия: ранее разработанное ПО (*previously developed software, PDS*) и готовое коммерческое ПО (*commercial-off-the-shelf software, COTS*). Эти понятия, а также *повторное использование* и *компонент ПО*, взаимосвязаны. Готовое коммерческое ПО является подмножеством ранее разработанного ПО, а само повторное использование обычно опирается на его компоненты. Возможно, несколько определений прояснят эту связь:

- *Повторное использование программного обеспечения:* существуют очень разные мнения о том, что считать таким повторным использованием, и разные определения этого термина. Я предпочитаю рассматривать его как процесс реализации или обновления систем с использованием существующих программных активов. Активами могут быть программные компоненты, требования, проект, исходный код и другие данные ЖЦ ПО, включая планы, стандарты, примеры и процедуры верификации, а также данные по квалификации инструмента. Такое повторное использование может происходить внутри

существующей системы, между схожими системами или между сильно различающимися системами.

- *Компонент ПО*: DO-178C определяет *компонент* так: «автономная часть, комбинация частей, подборок или единиц, выполняющая отдельную функцию системы» [3]. Однако я предпочитаю следующее, более описательное определение: «атомарный программный элемент, который может быть повторно использован или использован совместно с другими компонентами; в идеале он должен работать без модификации и без необходимости для инженера знать содержимое и внутреннюю функцию компонента. Однако интерфейс, функциональность, предусловия и постусловия, характеристики производительности и требуемые вспомогательные элементы должны быть хорошо известны» [4].
- *Готовое коммерческое ПО*: коммерчески доступные компоненты ПО, которые не предназначены для доработки или расширения пользователем, хотя могут конфигурироваться под его конкретные нужды [3].
- *Ранее разработанное ПО*: «ПО, уже разработанное для применения. Это понятие охватывает широкий спектр: от готовых коммерческих компонентов до программного обеспечения, разработанного в соответствии с прежними или текущими инструктивными материалами по ПО» [3].

Вместе эти определения показывают, что повторное использование программного обеспечения часто опирается на ранее разработанное ПО, упакованное в виде компонента. Оно может быть коммерчески доступным, то есть готовым коммерческим ПО, может быть разработано внутри компании в рамках прежнего проекта, по DO-178[],[*] по какому-либо другому руководству или стандарту, например военному или автомобильному, либо вообще без какого-либо руководства. В таблице 24.1 приведены некоторые примеры ранее разработанного ПО по четырем категориям. Часть такого программного обеспечения используется без изменений, а часть требует модификации для применения в новой системе или новой среде.

Таблица 24.1 Примеры ранее разработанного ПО

Не готовое коммерческое		
	ПО	Готовое коммерческое ПО
По DO-178[]	– приложение авионики, разработанное по DO-178A.- программное обеспечение управления полетом, соответствующее DO-178B, которое предстоит модифицировать для использования в схожей системе.- программа управления аккумуляторной батареей, соответствующая DO-178C, которая должна быть установлена на новом воздушном судне.	– операционная система реального времени (real-time operating system, RTOS), для которой доступны данные по DO-178B или DO-178C.- пакет поддержки платы, соответствующий DO-178C, для конкретной операционной системы реального времени и микропроцессора.
Не по DO-178[]	– программа электрической системы питания, разработанная по военному стандарту 498 Министерства обороны США.- программа системы управления полетом для военного самолета, разработанная по британскому стандарту Defence Standard 00-55.- программа тормозной системы автомобиля.- драйвер устройства для шины данных сети контроллеров (controller area network, CAN).	– операционная система без данных по DO-178[] (например, Windows).- библиотека, поставляемая вместе с компилятором.- программное обеспечение шины данных, используемое в автомобильной отрасли.- коммуникационный стек, соответствующий протоколу взаимодействия открытых систем (Open Systems Interconnection, OSI).

Чтобы программное обеспечение действительно было повторно использовано, а не просто извлечено из старого проекта, оно должно быть спроектировано как пригодное для повтор-

ного использования. Хотя никакой авиационный стандарт этого не *требует*, на практике это реальность и хорошая инженерная практика. Поэтому в этой главе обсуждается, как планировать и проектировать с расчетом на повторное использование. Затем внимание смещается к рассмотрению способов повторного использования ранее разработанного ПО. Завершает главу краткий обзор истории эксплуатации изделия, тесно связанной с ранее разработанным ПО.

24.2 Проектирование повторно используемых компонентов

Повторное использование программного обеспечения не возникает само собой. Успешная работа по повторному использованию требует планирования и аккуратных проектных решений. Чтобы компоненты можно было успешно использовать повторно, они должны быть спроектированы как пригодные для повторного использования. Проектирование с расчетом на повторное использование обычно увеличивает первоначальные сроки и стоимость разработки, но эти вложения окупаются, когда компонент повторно используется без существенной переделки.

Случаи Ariane 5, Therac-25 и другие ранее рассмотренные примеры позволяют кратко увидеть ситуации, в которых повторное использование программного обеспечения не было тщательно оценено и реализовано. По мере усложнения программного обеспечения и расширения областей его применения возрастают и проблемы его повторного использования в системах, критичных по безопасности. Повторное использование может быть жизнеспособным вариантом, однако оценивать и внедрять его нужно осторожно. Если программное обеспечение с самого начала проектируется как пригодное для повторного использования, это может помочь избежать подобных инцидентов.

В целом программная индустрия, особенно авиационная программная индустрия, все еще довольно незрела в вопросах разработки с расчетом на повторное использование. За десять лет, прошедших с тех пор, как я завершила магистерскую диссертацию по этой теме и пришла к тому же выводу, в области повторного использования появились определенные подвижки для таких компонентов, как операционные системы реального времени и библиотечные функции. К сожалению, проектирование с расчетом на повторное использование по-прежнему остается в авиационной отрасли довольно ускользающей целью. Часто проект начинается с намерением спроектировать решение так, чтобы его можно было использовать повторно. Однако как только начинают действовать ограничения по срокам и бюджету, цели повторного использования забываются.

Ниже приведены 16 рекомендаций, которые стоит учитывать при проектировании с расчетом на повторное использование. Эти взаимосвязанные рекомендации суммируют важнейшие организационные и технические принципы эффективного повторного использования.

Рекомендация 1: Добейтесь долгосрочной приверженности руководства повторному использованию. Проектирование компонента как пригодного для повторного использования займет больше времени и будет стоить дороже, чем проектирование компонента без такого расчета. Насколько именно вырастут сроки и стоимость, зависит от используемых процессов, опыта и предметной компетентности организации, типа разрабатываемого компонента, приверженности руководства и множества других факторов. По некоторым оценкам, повторно используемый компонент обходится в два-три раза дороже, чем тот же компонент для разового применения [5]. Руководство должно понимать эту реальность и брать на себя долгосрочное обязательство. Чтобы повторное использование было успешным, его должно поддерживать высшее руководство. Большинство неудач в этой области происходит из-за того, что руководству не хватает решимости поддерживать усилия тогда, когда становится тяжело.

Рекомендация 2: Создайте команду по повторному использованию и обучите ее. Хорошо подготовленная команда, для которой разработка повторно используемого программного обеспечения является основной целью, добивается большего успеха, чем команда, работающая как обычно. Это может быть небольшая выделенная команда, а может быть группа людей, которые уделяют этой работе только часть своего времени. Однако для обеспечения ответственности важно, чтобы команда была явно определена и чтобы все ее участники были должным образом обучены.

Рекомендация 3: Проанализируйте существующие проекты, чтобы выявить проблемы, мешающие повторному использованию. Полезно проанализировать существующие программные проекты внутри компании, чтобы понять, какие именно проблемы мешают повторному использованию. Этому могут препятствовать проектные практики, практики кодирования, аппаратные интерфейсы, среда разработки и особенности компилятора. Составив перечень проблем, мешающих повторному использованию, команда может начать вырабатывать стратегии их преодоления.

Рекомендация 4: Попробуйте пилотный проект. Вместо того чтобы пытаться перестроить всю организацию за одну ночь, часто лучше начать с небольшого пилотного проекта. Такой проект может стать основой для выявления практик повторного использования и обучения команды. Небольшой успешный проект нарабатывает опыт и уверенность.

Рекомендация 5: Документируйте практики повторного использования и извлеченные уроки. Прежде чем запускать разработку повторно используемого компонента, подготовьте практики, которым должна следовать команда. После первого или второго проекта эти практики следует обновить. В идеале практики и процедуры постоянно уточняются по мере накопления уроков из реального опыта. Со временем такие практики могут стать общеор-

ганизационными рекомендациями или процедурами.

Рекомендация 6: Определяйте предполагаемых пользователей на этапе разработки компонента и поддерживайте их на всем протяжении работы. Поскольку пользователи являются заказчиками компонента, важно определить, кто именно будет его использовать, чтобы убедиться, что их потребности удовлетворены, возможные противоречащие друг другу потребности выявлены, а сами пользователи уведомлены о любых проблемах, которые могут возникнуть при разработке и сопровождении компонента. Также важно спроектировать компонент удобным для пользователя. Это включает такие характеристики, как: (1) легкость идентификации, то есть пользователи должны без труда понимать, отвечает ли компонент их потребностям; (2) легкость использования, то есть пользователи должны быстро осваивать работу с компонентом; и (3) пригодность для использования интеграторами с самым разным уровнем опыта, от новичков до экспертов [6].

Рекомендация 7: Внедрите процесс доменной инженерии. Надежный процесс доменной инженерии крайне важен для успешного повторного использования программного обеспечения. Домен – это «группа или семейство связанных систем. Все системы в этом домене разделяют некоторый набор возможностей и/или данных» [7]. Доменная инженерия – это процесс создания активов через анализ, проектирование и реализацию домена, чтобы ими можно было управлять и использовать их повторно. Доменный инженер предлагает архитектуру, которая удовлетворяет большинство потребностей приложений и пригодна для последующего повторного использования. Важно различать понятия *доменная инженерия* и *инженерия повторного использования*. *Доменная инженерия* означает разработку для повторного использования, тогда как *инженерия повторного использования* означает разработку с использованием уже пригодных для повторного использования решений. Качественная доменная инженерия критически важна для того, чтобы можно было повторно использовать компоненты целиком или проектные части этих компонентов, а не просто спасать код из прошлого.

Рекомендация 8: Определите и документируйте домен использования. Домен использования – это объявленный набор характеристик, для которого можно показать следующее:

- Компонент соответствует своим функциональным требованиям, требованиям к производительности и требованиям безопасности.
- Компонент удовлетворяет всем утверждениям и гарантиям относительно определенных для него распределяемых ресурсов и возможностей.
- Производительность компонента полностью охарактеризована, включая обработку отказов и ошибок, режимы отказа и поведение при неблагоприятных воздействиях среды [8].

Важно документировать домен использования, поскольку при повторном использовании компонента в новой установке потребуется выполнить анализ домена использования, чтобы оценить возможность такого повторного использования. Анализ домена использования оценивает последующее повторное использование компонента и обеспечивает, что: (1) учтены все допущения, сделанные разработчиком компонента; (2) рассмотрено влияние компонента на установку; (3) рассмотрено влияние новой среды на компонент; и (4) любые интерфейсы с компонентом в новой установке согласуются с доменом использования компонента и спецификацией его интерфейса [8].

Рекомендация 9: Создавайте небольшие, четко определенные компоненты. Стив Макконнелл пишет: «Возможно, вам лучше сосредоточить усилия по повторному использованию на создании небольших, точных, специализированных компонентов, а не больших, громоздких, универсальных компонентов. Разработчики, которые пытаются создавать повторно используемое программное обеспечение путем создания универсальных компонентов, редко способны адекватно предугадать потребности будущих пользователей... “Большой и громоздкий” означает “слишком трудно понять”, а это означает “слишком подвержен ошибкам при использовании”» [5]. Чтобы компонент можно было использовать повторно, его функциональность должна быть ясной и хорошо документированной. Функциональность определяется на высоком уровне тем, *что* делает компонент, а не *как* он реализован. У функциональности должна быть одна цель, она должна предоставлять только функции, связанные с этой целью, и иметь разумный размер, то есть быть не слишком маленькой для практического применения и не слишком большой, чтобы не стать неуправляемой. Кроме того, чтобы обеспечить плавную и успешную интеграцию, компонент должен иметь хорошо определенный интерфейс. Интерфейс определяет, как пользователь взаимодействует с компонентом. Удачный интерфейс имеет согласованный синтаксис, логичную структуру, предсказуемое поведение и последовательный способ обработки ошибок. Хорошо определенный интерфейс является полным, согласованным и целостным: он предоставляет пользователю все необходимое, чтобы компонент заработал [9].

Рекомендация 10: Проектируйте с учетом переносимости. Переносимость является желательным свойством большинства программных продуктов, поскольку увеличивает ценность программного пакета как за счет продления срока его полезного использования, так и за счет расширения набора установок, в которых он может применяться [10]. Существуют два типа переносимости: *двоичная переносимость*, то есть перенос исполняемой формы, и *переносимость исходного кода*, то есть перенос представления на исходном языке. Двоичная переносимость явно желательна, но обычно возможна только между сильно схожими процессорами, например с одним и тем же набором двоичных инструкций, либо если доступен двоичный транслятор. Переносимость исходного кода предполагает наличие исходного

кода, но дает возможности адаптировать программный модуль к широкому спектру сред [11]. Чтобы добиться двоичной переносимости или переносимости исходного кода, программное обеспечение должно проектироваться с учетом переносимости. В общем случае это требует таких проектных стратегий [10]:

1. Определить минимально необходимые требования и допущения по среде.
2. Исключить ненужные допущения на всем протяжении проектирования.
3. Определить требуемые интерфейсы, специфичные для среды, например вызовы процедур, параметры и структуры данных.
4. Для каждого интерфейса сделать одно из следующего:
 - a. Инкапсулировать интерфейс в подходящий модуль, пакет, объект и так далее, то есть заранее предусмотреть необходимость адаптации интерфейса для каждой целевой системы.
 - b. Определить стандарт для интерфейса, который будет доступен в большинстве целевых сред. И следовать этому стандарту на всем протяжении проектирования.

Рекомендация 11: Проектируйте компонент робастным. Бертран Мейер пишет: «Компонент не должен отказывать, если он используется правильно» [6]. Поскольку у компонента будет несколько пользователей, его необходимо проектировать робастным. Компонент должен предусматривать неожиданные входные данные и обрабатывать их, например с использованием механизмов обработки ошибок. Робастность необходимо учитывать при разработке требований к компоненту и его проекта.

Рекомендация 12: Упаковывайте данные так, чтобы они были пригодны для повторного использования. В главе 12 приказа FAA Order 8110.49 обсуждается повторное использование данных ЖЦ ПО в среде сертификации воздушных судов [18]. Чтобы данные, как и само программное обеспечение, были пригодны для повторного использования, его необходимо упаковывать именно с таким расчетом. Часто это означает наличие полного набора данных ЖЦ ПО DO-178C для каждого компонента. Order 8110.49 дает рекомендации по повторному использованию внутри компании, а консультативный циркуляр FAA AC 20-148 рассматривает концепцию повторного использования программных компонентов за пределами границ одной компании, например повторное использование операционной системы реального времени в нескольких системах авионики. AC 20-148 содержит рекомендации о том, как документировать компонент, чтобы он был пригоден для повторного использования. Независимо от того, запрашивается ли письмо Федерального управления гражданской авиации США (Federal Aviation Administration, FAA) о принятии повторного использования, компонент все равно можно упаковать так, чтобы он был пригоден для повторного

использования от программы к программе. И приказ, и циркуляр содержат рекомендации по такой упаковке.

Рекомендация 13: Хорошо документируйте повторно используемый компонент. Поскольку часто заранее неизвестно, кто будет использовать компонент в будущем, важно документировать его максимально полно. Это включает подготовку документации, обеспечивающей правильное использование и интеграцию компонента, например спецификации интерфейса и руководства пользователя, а также данных для поддержки сертификации и сопровождения. АС 20-148 требует создания информационного листа, в котором объясняются сведения, необходимые пользователю повторно используемого компонента ПО для обеспечения правильного использования компонента. Раздел 6.i АС 20-148 требует включать в такой лист следующие данные: функции компонента, ограничения, анализ потенциальных проблем безопасности интерфейсов, допущения, конфигурацию, сопровождающие данные, открытые сообщения о проблемах, характеристики компонента, например наихудшее время выполнения, объем памяти и пропускную способность, а также другую значимую информацию, поддерживающую использование компонента [12].

Рекомендация 14: Документируйте обоснование проектных решений. Чтобы эффективно повторно использовать компонент ПО, необходимо хорошо документировать обоснование проектных решений. Важны и проектные решения, относящиеся к самому компоненту, и проектные решения системы, которая первой интегрирует этот компонент. Задокументированные проектные решения помогают определить, где компонент можно «уместно и с выгодой повторно использовать» [13]. Кроме того, задокументированные проектные решения для системы, использующей компонент, также могут помочь определить, подходит ли компонент для повторного использования в другой системе. Дин Аллеманг делит обоснование проектных решений на две категории: (1) внутреннее обоснование, то есть связь частей проекта с другими частями того же проекта, иными словами способ взаимодействия компонента с другими компонентами; и (2) внешнее обоснование, то есть связь частей проекта с частями других проектов, то есть использование компонента в нескольких системах [13].

Рекомендация 15: Документируйте информацию по безопасности для пользователя. При разработке компонента и его последующем повторном использовании необходимо учитывать вопросы безопасности. Крайне важно, чтобы разработчик компонента определил условия отказа, функции безопасности, механизмы защиты, архитектуру, ограничения, уровни программного обеспечения, спецификации интерфейсов и предполагаемое использование компонента. Все интерфейсы и настраиваемые параметры должны быть проанализированы, чтобы описать функциональные последствия и последствия для производительности этих параметров и интерфейсов для пользователя. Анализ документирует действия, необходимые со стороны пользователя для обеспечения правильной работы. Кроме того, согласно

разделу 5.f АС 20-148, разработчик повторно используемого компонента ПО должен «подготовить анализ поведения такого компонента, которое может неблагоприятно повлиять на реализацию у пользователей, например в части уязвимостей, требований к обособлению, последствий отказов аппаратуры, требований к резервированию, задержек данных и проектных ограничений для его корректной работы. Этот анализ может поддерживать анализ безопасности, выполняемый интегратором или заявителем» [12].

Рекомендация 16: Делайте упор на качество, а не на количество. Поскольку повторно используемые компоненты могут иметь нескольких пользователей, важно обеспечить, чтобы каждый компонент работал как требуется. Макконнелл пишет: «Успешное повторное использование требует создания компонентов, практически свободных от ошибок. Если разработчик, пытающийся применить повторно используемый компонент, обнаружит, что он содержит дефекты, программа повторного использования быстро потеряет привлекательность. Программа повторного использования, основанная на низкокачественных программах, может фактически увеличить стоимость разработки программного обеспечения... Если вы хотите внедрить повторное использование, сосредоточьтесь на качестве, а не на количестве» [5]. При разработке АС 20-148 Федеральное управление гражданской авиации США подчеркивало тот факт, что первое одобрение повторно используемого компонента требует высокого уровня надзора и вовлеченности сертифицирующего органа, чтобы убедиться в его корректном функционировании.

24.3 Повторное использование ранее разработанного ПО

Рассматриваются три аспекта этой темы: (1) процесс оценки ранее разработанного ПО для включения в систему, критичную по безопасности, особенно когда требуется соответствие нормам гражданской авиации, (2) специальные соображения при повторном использовании программного обеспечения, которое не разрабатывалось по DO-178[],[*] и (3) дополнительные факторы, которые нужно оценивать, если оно одновременно является готовым коммерческим ПО.

24.3.1 Оценка ранее разработанного ПО для использования в изделиях гражданской авиации

Для лучшего понимания процесса оценки применения ранее разработанного ПО в изделии гражданской авиации, то есть в воздушном судне, двигателе или воздушном винте, приводится блок-схема, см. рисунок 24.1. Хотя она, возможно, не охватывает все возможные ситуации, она должна охватывать большинство тех, что относятся к проектам гражданской авиации. Подобный подход можно применить и в других предметных областях, хотя там он может потребовать некоторых изменений. Каждый блок на блок-схеме пронумерован и описывается ниже:

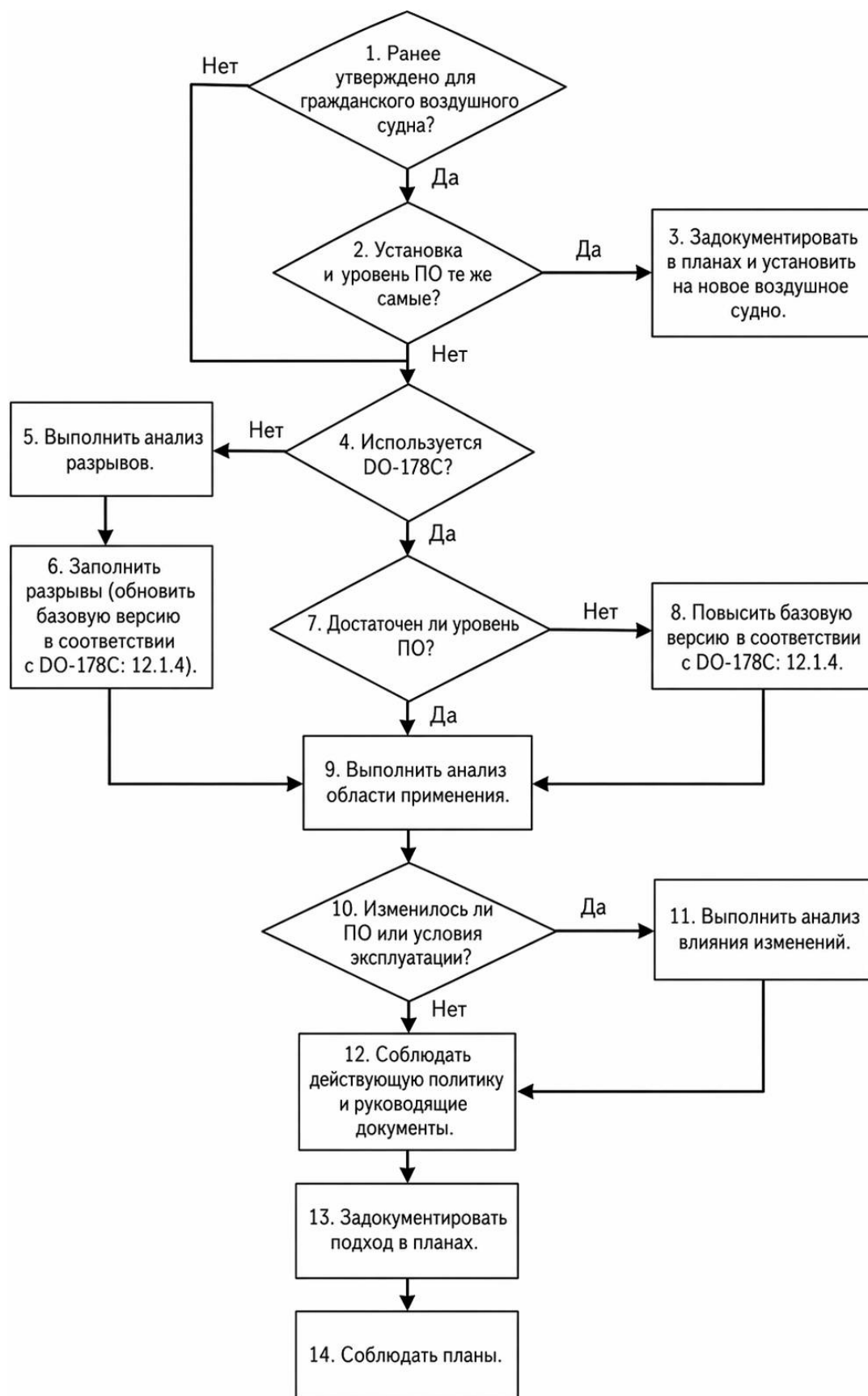


Рисунок 24.1 Процесс оценки ранее разработанного ПО для проекта гражданской авиации.

1. *Определите, было ли ранее разработанное ПО одобрено в установке на гражданском воздушном судне. Если ранее разработанное ПО ранее уже было одобрено в сертифицированном гражданском воздушном судне, двигателе или воздушном винте, оно может*

быть пригодным для повторного использования без доработки и повторной верификации. Если же оно ранее не одобрялось, потребуется показать, что оно соответствует DO-178C или эквивалентному уровню гарантии.

2. *Определите, будут ли установка и уровень ПО теми же самыми.* Если программное обеспечение устанавливается в ту же самую систему и используется тем же самым образом, например система предупреждения столкновений и предотвращения столкновений устанавливается на другом воздушном судне без изменения аппаратуры или самого программного обеспечения, и влияние на безопасность остается тем же, то ранее разработанное ПО, вероятно, пригодно для повторного использования без доработки и повторной верификации. Обычно на системном уровне требуется анализ сходства, чтобы оценить применение программного обеспечения, подтвердить соображения безопасности и обосновать утверждение о повторном использовании. Эта информация, скорее всего, войдет в планы системного уровня. Если установка меняется или уровень ПО недостаточен, требуется дальнейшая оценка, см. блок 4.
3. *Задokumentируйте намерение использовать ранее разработанное ПО в планах.* Намерение использовать его и результаты анализа сходства должны быть задокументированы в планах. В зависимости от ситуации это могут быть планы системного уровня, например если повторно используется вся система, либо План сертификации ПО (PSAC), если повторно используется только программный компонент. Планы должны объяснять эквивалентность установки и достаточность уровня ПО, как указано в пунктах 1 и 2.
4. *Было ли программное обеспечение первоначально одобрено по DO-178[]?* Назначение этого блока процесса состоит в том, чтобы определить, соблюдались ли DO-178C или его предшественники при первоначальной разработке. Если нет, выполняется анализ разрывов, чтобы выявить недостающее, и определяется подход к восполнению этих разрывов, см. блоки 5 и 6.
5. *Выполните анализ разрывов.* Анализ разрывов включает оценку ранее разработанного ПО и сопровождающих его данных на соответствие целям DO-178C. Для ПО неизвестного происхождения (SOUP) такой анализ может оказаться невозможным, поскольку данные обычно недоступны. Такое программное обеспечение обычно может быть одобрено только до уровня D или E.
6. *Закройте разрывы.* После выявления разрывов их необходимо закрыть. Раздел 12.1.4 DO-178C, «Модернизация базовой версии разработки», дает рекомендации по выполнению этой задачи. В зависимости от исходной разработки это может потребовать значительных усилий и заставить пересмотреть сам план повторного использования. Неко-

торые альтернативные подходы, которые могут применяться для закрытия разрывов, рассматриваются далее в этой главе, см. раздел 24.3.2. Если ранее разработанное ПО является готовым коммерческим компонентом, см. раздел 24.3.3, где приведены дополнительные рекомендации по выявлению и закрытию разрывов.

7. *Определите, достаточен ли уровень ПО.* Определите, какой уровень ПО требуется для новой установки, и достаточен ли уровень, присвоенный ранее разработанному ПО. Если программное обеспечение не разрабатывалось по DO-178[],[*] ответ будет *НЕТ*, и потребуется анализ разрывов, см. блоки 5 и 6. Если оно разрабатывалось по DO-178 или DO-178А, эквивалентные уровни для DO-178В и DO-178С приведены в таблице 24.2.

Таблица 24.2 Эквивалентность уровней ПО по DO-178, DO-178А, DO-178В и DO-178С

Уровень ПО по DO-178В/С, требуемый для новой установки	Унаследованное ПО уровня Critical/Level 1 по DO-178 или DO-178А	Унаследованное ПО уровня Essential/Level 2 по DO-178 или DO-178А	Унаследованное ПО уровня Nonessential/Level 3 по DO-178 или DO-178А
А	Да, но требуется анализ	Нет	Нет
В	Да	Нет, требуется анализ	Нет
С	Да	Да	Нет
Д	Да	Да	Нет
Е	Да	Да	Да

Источник: Federal Aviation Administration, Software Approval Guidelines, Order 8110.49, Change 1, September 2011.

8. *Модернизируйте базовую версию разработки.* Если присвоенный ранее разработанному ПО уровень недостаточен, данные и, возможно, само программное обеспечение потребуется обновить так, чтобы закрыть дополнительные цели DO-178[]. Раздел 12.1.4 DO-178С описывает этот процесс. Обычно он включает дополнительные мероприятия верификации. Если оно не разрабатывалось по DO-178В или DO-178С, такая модернизация может также потребовать анализа разрывов, см. блоки 5 и 6, чтобы довести его до соответствия DO-178С. Необходимость такого анализа зависит от желаемого повышения уровня ПО и от способности существующих данных поддерживать потребности безопасности; этот вопрос следует тесно координировать с сертифицирующим органом.

9. *Выполните анализ домена использования.* Анализ домена использования оценивает, совместимы ли новая установка и новая среда с ограничениями и допущениями ранее разработанного ПО. В контексте гражданской авиации анализ домена использования должен определить следующее:

- a. Используемое программное обеспечение совпадает с первоначально одобренной версией.
- b. Отсутствуют неблагоприятные последствия для безопасности или эксплуатационных возможностей воздушного судна.
- c. Характеристики работы оборудования остаются теми же.
- d. Все открытые сообщения о проблемах оценены с точки зрения влияния в новой установке. Если сообщения о проблемах недоступны, это может сделать повторное использование ранее разработанного ПО невозможным.
- e. Диапазоны, типы данных и параметры эквивалентны первоначальной установке.
- f. Интерфейсы остаются теми же.

Часть этого анализа уже рассматривается в других блоках, однако назначение блока 9 состоит в том, чтобы собрать информацию в одном месте и более полно оценить предполагаемое повторное использование. Такой анализ может выявить необходимость изменений в программном обеспечении.

Если ранее разработанное ПО будет модифицироваться, этот анализ можно выполнять совместно с анализом влияния изменений и документировать как часть анализа влияния изменений. Однако если ни программное обеспечение, ни его среда не меняются, этот анализ домена использования включается в планы, обычно в План сертификации ПО (PSAC) или в системный план сертификации.

10. *Определите, изменились ли программное обеспечение или его среда.* Анализ домена использования, см. блок 9, применяется для определения того, есть ли какие-либо изменения в установке ранее разработанного ПО. Кроме того, если изменилась среда разработки, например компилятор, его настройки, версия генератора кода или версия компоновщика, либо изменилось само программное обеспечение, необходим анализ влияния изменений, а соответствующие данные ЖЦ ПО потребуются скорректировать. Разделы 12.1.2, 12.1.3 DO-178C рассматривают изменения установки, применения и среды разработки.

11. *Выполните анализ влияния изменений.* Как отмечалось в главе 10, модифицированное программное обеспечение требует анализа влияния изменений, чтобы оценить последствия изменения и спланировать повторную верификацию.[*] Как также обсуждалось

в главе 10, анализ влияния изменений используется для определения того, является ли изменение программного обеспечения *существенным* или *несущественным*. Если ранее разработанное ПО классифицируется как *унаследованное*, то есть было создано по более ранней версии DO-178, чем требуется сертификационным базисом, это может повлиять на способ реализации изменения. Согласно Order 8110.49, для *несущественных изменений* можно использовать процесс, который применялся при первоначальной разработке унаследованного ПО: например, несущественное изменение ранее разработанного ПО, соответствующего DO-178A, может выполняться по процессу DO-178A. Однако если изменение классифицируется как *существенное*, то это изменение и все последующие изменения того же ранее разработанного ПО должны выполняться по версии DO-178, требуемой сертификационным базисом, вероятнее всего по DO-178C для новых проектов.[*]

12. *Следуйте применимой сертификационной политике и руководящим материалам.* Помимо рекомендаций раздела 12.1 DO-178C, большинство сертифицирующих органов имеют политику или руководящие материалы, относящиеся к ранее разработанному ПО и/или повторному использованию, например FAA Order 8110.49 и AC 20-148. Кроме того, могут выпускаться проектные issue papers (документы по спорным вопросам сертификации) или их аналоги, если существующие рекомендации не подходят для конкретного сценария проекта.
13. *Задokumentируйте подход в планах и получите согласие сертифицирующего органа.* План использования ранее разработанного ПО должен быть задokumentирован в PSAC или, возможно, в системном плане сертификации. В нем описываются функции этого программного обеспечения, его исходное происхождение, любые изменения установки, самого продукта или среды разработки, а также результаты анализа влияния изменений и (или) анализа домена использования. Если оно не разрабатывалось по DO-178[], следует описать детали анализа разрывов и альтернативного подхода или подходов. В PSAC также следует объяснить все необходимые изменения и способ их верификации. Кроме того, в нем описывается, как будут учтены вопросы управления конфигурацией ПО и гарантии качества ПО в соответствии с разделами 12.1.5[†] и 12.1.6[‡] DO-178C. После завершения PSAC передается сертифицирующему органу. Обычно большинство сертифицирующих органов также хотят видеть анализ влияния изменений. Как отмечалось в главе 10, анализ влияния изменений часто включается в PSAC.

Важно получить согласие сертифицирующего органа, прежде чем заходить слишком далеко в усилиях по повторному использованию.

14. *Следуйте планам и реализуйте решение в соответствии с согласованным процессом.*

После того как сертифицирующий орган одобрит планы, а на это может потребоваться несколько итераций, утвержденным планам необходимо следовать. Следует выполнить все необходимые доработки, верификацию и документирование. По мере развития проекта анализ влияния изменений и планы могут потребовать обновления.

24.3.2 Повторное использование ранее разработанного ПО, не разработанного по DO-178[]

Как отмечено на рисунке 24.1 в блоках 5 и 6, если программное обеспечение не разрабатывалось по DO-178[], выполняется анализ разрывов, чтобы выявить цели DO-178C, которые не выполнены. В зависимости от исходной разработки закрытие этих разрывов может оказаться тривиальной задачей, а может потребовать чрезвычайно больших трудозатрат. Для ПО неизвестного происхождения (SOUP) может оказаться эффективнее начать заново, чем пытаться восстановить несуществующие данные. Для ранее разработанного ПО, по которому доступны хотя бы некоторые данные, восполнение разрывов может быть жизнеспособным путем. Ранее разработанное ПО с очень хорошей историей эксплуатации, но не в гражданской авиации, например готовый коммерческий компонент, такой как операционная система реального времени, военное приложение или автомобильный компонент, может успешно пройти по этому пути.

Анализ разрывов сопоставляет имеющиеся данные с тем, что требуется по DO-178C, чтобы определить объем дополнительной работы. По моему опыту, для программного обеспечения уровней A и B разрыв обычно оказывается довольно широким. Если исходный код недоступен, обычно максимумом является уровень D.[*] Если данных доступно очень мало, но исходный код есть, наиболее распространенными вариантами становятся история эксплуатации и обратная разработка.

Раздел 12.4 DO-278A и раздел 4.5 DO-248C описывают типовые альтернативные подходы, применяемые для того, чтобы обеспечить для ранее разработанного ПО тот же уровень доверия, который был бы достигнут, если бы цели DO-178C изначально были выполнены. В таблице 24.3 суммированы некоторые из наиболее распространенных подходов. В большинстве сценариев разрывы такого ПО закрываются комбинацией этих подходов. История эксплуатации рассматривается далее в этой главе, а обратная разработка разбирается в главе 25.

Таблица 24.3 Сводка альтернативных подходов для ранее разработанного ПО

Подход	Описание	Ограничения
История эксплуатации изделия	DO-178C определяет историю эксплуатации как непрерывный период времени, в течение которого программное обеспечение эксплуатируется в известной среде и в течение которого регистрируются последовательные отказы. Подход подробно рассматривается далее в главе.	Среда, в которой накоплена история, может не совпадать с предполагаемой новой средой; трудно доказать достаточность регистрации сообщений о проблемах и сбора данных; ранее разработанное ПО могло многократно меняться, из-за чего сложно доказать целостность истории; наработки может оказаться недостаточно; подход субъективен и его трудно согласовать с сертифицирующим органом без дополнительных аргументов.
Признание процесса	Если ранее разработанное ПО было одобрено или аккредитовано по определенному процессу, можно попытаться зачесть часть или все цели DO-178. В качестве примеров приводятся Military Standard 498, DOD-STD-2167A и процесс с оценкой Capability Maturity Model Integration института Software Engineering Institute.	Сам процесс обычно не покрывает все цели DO-178C; данные, необходимые для оценки такого процесса пользователем, могут быть недоступны.
Обратная разработка	Обратная разработка понимается как создание данных более высокого уровня из уже существующих программных данных. Примеры: получение исходного кода из объектного или исполняемого объектного кода, а также восстановление требований высокого уровня из требований низкого уровня.	Это субъективный подход, который сертифицирующие органы не приветствуют; трудно надежно перейти от низкого уровня абстракции к высокому, поэтому восстановленные требования часто повторяют код; исходный замысел разработчиков может быть утрачен; для качественной обратной разработки нужна очень опытная команда.
Ограничение функциональности	Использование ранее разработанного ПО ограничивается подмножеством общей функциональности. Для этого можно применять проверки во время выполнения, отключенный код, встроенные ограничения или обертки. Такой подход часто используется в авиационных ОСРВ, где оставляют только безопасное подмножество функций.	Трудно доказать, что нежелательный код никогда не будет выполнен; защитные механизмы могут потребовать дополнительной верификации.
Архитектурное смягчение последствий	Система проектируется так, чтобы защититься от нежелательного поведения программного обеспечения. В качестве типичных средств названы обособление и мониторинг безопасности. Обособление изолирует взаимодействие с остальными программными средствами, а мониторы помогают обнаруживать потенциальные отказы.	Без знания всей функциональности ранее разработанного ПО может быть трудно доказать, что все нежелательные эффекты действительно нейтрализованы.
Дополнительное тестирование	Выполняется расширенное тестирование, чтобы убедиться, что программное обеспечение работает как задумано и не вызывает нежелательных эффектов. В качестве примеров приведены исчерпывающие входные тесты для простой функции с изолированными входами и выходами, расширенное тестирование на робастность для проверки реакции на выход за диапазон, нештатные и ошибочные входы, системные испытания на уровне системы, длительные испытания с непрерывной работой на широком наборе нормальных и аномальных входов без сброса системы, а также использование системы в учебных целях с участием множества независимых пользователей.	Трудно доказать полное отсутствие неумышленной функциональности, особенно в крупном или сложном ПО.

Источник: RTCA DO-278A и RTCA DO-248C.

24.3.3 Дополнительные соображения по готовому коммерческому ПО

Как отмечалось ранее, готовое коммерческое ПО представляет собой особый вид ранее разработанного ПО. Примеры таких компонентов приведены в таблице 24.1. Раздел 12.4 DO-278A содержит ряд конкретных рекомендаций по его приобретению и интеграции в си-

стему, критичную по безопасности. DO-178C не содержит эквивалентных рекомендаций, однако подход DO-278A можно применять и в проекте по DO-178C. DO-278A продвигает концепцию обоснования доверия к целостности готового коммерческого ПО, которое разрабатывается для того, чтобы показать: такой компонент обеспечивает тот же уровень доверия, что и программное обеспечение, разработанное по DO-278A или DO-178C. Обоснование доверия к целостности для соответствия DO-178C включает следующую информацию [14]:

- Утверждения о целостности готового коммерческого ПО и указание, какие цели DO-178C этим обоснованием не покрываются.
- Среду, в которой будет использоваться готовое коммерческое ПО.
- Требования, которым удовлетворяет готовое коммерческое ПО.
- Выявление, оценку и смягчение последствий ненужных возможностей готового коммерческого ПО.
- Объяснение того, какие цели DO-178C удовлетворяются существующими данными жизненного цикла готового коммерческого ПО.
- Выявление целей DO-178C, которые не удовлетворены, и объяснение того, как будет достигнут эквивалентный уровень доверия с использованием альтернативных подходов. Описание стратегий, которые будут применяться.
- Перечень всех данных, поддерживающих это обоснование, включая дополнительные данные ЖЦ ПО, которые будут созданы альтернативными методами.
- Список допущений и обоснований, используемых в аргументации по целостности. Описание процессов проверки того, что все непокрытые цели, выявленные анализом разрывов, выполнены.
- Свидетельства того, что готовое коммерческое ПО обладает той же целостностью, как если бы все цели были выполнены в ходе первоначальной разработки.

Кроме того, при определении осуществимости применения готового коммерческого ПО в системе, критичной по безопасности, необходимо рассмотреть следующие специфические для готового коммерческого ПО вопросы:

1. Доступность подтверждающих данных ЖЦ ПО.
2. Пригодность готового коммерческого ПО для использования в критичных по безопасности системах. Многие компоненты готового коммерческого ПО не проектировались с учетом требований безопасности.
3. Стабильность готового коммерческого ПО. Следует оценить следующие факторы:

- a. Как часто оно обновлялось?
 - b. Выпускались ли исправления?
 - c. Предоставляется ли или доступен ли полный набор данных ЖЦ ПО для каждого обновления?
 - d. Будет ли поставщик уведомлять о выходе обновлений, объяснять причины их появления, предоставлять сообщения о проблемах, которые были исправлены, чтобы поддержать анализ влияния изменений, и предоставлять обновленные данные для сопровождения изменения?
4. Техническая поддержка, доступная со стороны поставщика. Следует учитывать следующее:
- a. Если при использовании готового коммерческого ПО будут выявлены проблемы, готов ли поставщик и обязан ли он их исправлять?
 - b. Готов ли поставщик поддерживать сертификацию?
 - c. Готов ли поставщик и обязан ли он раскрывать известные проблемы на протяжении всего срока использования готового коммерческого ПО, например проблемы, выявленные другими пользователями?
 - d. Что произойдет, если поставщик прекратит работу или решит больше не поддерживать это программное обеспечение?
5. Управление конфигурацией готового коммерческого ПО. Следует учитывать следующее:
- a. Имеется ли система регистрации сообщений о проблемах?
 - b. Трассируются ли изменения до предыдущих базовых версий?
 - c. Имеют ли готовое коммерческое ПО и сопровождающие его данные ЖЦ ПО однозначную идентификацию?
 - d. Существует ли контролируемый процесс выпуска?
 - e. Предоставляются ли открытые сообщения о проблемах?
6. Защищенность готового коммерческого ПО от вирусов, уязвимостей безопасности и тому подобного.
7. Поддержка инструментов и аппаратуры для готового коммерческого ПО.
8. Совместимость готового коммерческого ПО с целевым вычислителем и взаимодействующими системами.

9. Возможность модификации или конфигурирования готового коммерческого ПО.
10. Возможность отключить или удалить нежелательную либо ненужную функциональность готового коммерческого ПО.
11. Достаточность гарантии качества у поставщика.

24.4 История эксплуатации изделия

В этом разделе кратко объясняется, что такое история эксплуатации изделия, и какие факторы следует учитывать, предлагая ее как альтернативный метод для ранее разработанного ПО.

24.4.1 Определение истории эксплуатации изделия

Как отмечено в таблице 24.3, DO-178C определяет *историю эксплуатации изделия* следующим образом: «Непрерывный период времени, в течение которого программное обеспечение эксплуатируется в известной среде и в течение которого регистрируются последовательные отказы» [3].

Это определение включает понятия регистрации сообщений о проблемах, среды, включая эксплуатационную среду и среду целевого вычислителя, а также времени.

И отрасль, и сертифицирующие органы затратили значительные усилия на то, чтобы найти практически применимый способ использовать историю эксплуатации изделия для получения сертификационного зачета. Группа сертифицирующих органов по ПО (Certification Authorities Software Team, CAST)[*] опубликовала документ по этой теме (CAST-1); Федеральное управление гражданской авиации США спонсировало исследования в этой области, результатом которых стали исследовательский отчет и руководство; одна страница текста в DO-178B превратилась в четыре страницы в DO-178C; а DO-248C включает семи-страничный дискуссионный документ по этой теме. Однако, несмотря на усилия по прояснению вопроса, применять этот подход по-прежнему трудно.

Раздел 12.3.4 DO-178C поясняет, что обоснование истории эксплуатации изделия зависит от следующих факторов [3]: (1) управления конфигурацией ПО, (2) эффективности мероприятий по регистрации сообщений о проблемах, (3) стабильности и зрелости программного обеспечения, (4) релевантности среды истории эксплуатации изделия, (5) продолжительности истории эксплуатации изделия, (6) фактических частот ошибок в истории эксплуатации изделия и (7) влияния модификаций. Далее этот раздел дает дополнительные рекомендации по релевантности истории эксплуатации, достаточности накопленной истории эксплуатации, сбору и анализу проблем, выявленных в течение истории эксплуатации, а также по информации, которую следует включать в План сертификации ПО (PSAC) при предложении истории эксплуатации как альтернативного метода.

История эксплуатации изделия может применяться к программному обеспечению, которое не разрабатывалось по DO-178[], а также к программному обеспечению, которое разрабатывалось по DO-178[], но на более низком уровне, чем требуется для желаемой системы более высокого уровня.

24.4.2 Трудности при попытке получить зачет на основе истории эксплуатации изделия

На сегодняшний день добиться успешного зачета, опираясь только на историю эксплуатации изделия, практически невозможно. Однако ее успешно использовали как дополнение к другим альтернативным подходам, например к обратной разработке или признанию процесса.

Как отмечается в исследовательском отчете Федерального управления гражданской авиации США под названием *Software Service History Report*, авторами которого являются Uma и Tom Ferrell, определение истории эксплуатации изделия в DO-178C «очень похоже на определение надежности IEEE [Institute for Electrical and Electronics Engineers, Институт инженеров электротехники и электроники], которое звучит так: “способность изделия выполнять требуемую функцию в заданных условиях в течение заданного периода времени”» [15].[†] Ни DO-178C, ни сертифицирующие органы не поощряют использование моделей надежности программного обеспечения, поскольку исторически их точность не была доказана. Из-за сходства между надежностью и историей эксплуатации также трудно построить удовлетворительное обоснование, используя одну только историю эксплуатации изделия.

Еще один фактор, из-за которого историю эксплуатации изделия трудно доказать, состоит в том, что данные, собранные в течение истории изделия, часто оказываются недостаточными. Компании обычно заранее не планируют заявлять зачет на основе истории эксплуатации изделия, поэтому механизм регистрации сообщений о проблемах может отсутствовать и данные просто не собираются.

24.4.3 Факторы, которые следует учитывать при заявлении зачета на основе истории эксплуатации изделия

При заявлении зачета на основе истории эксплуатации необходимо рассмотреть следующее:

1. Должна быть продемонстрирована релевантность истории эксплуатации. Согласно разделу 12.3.4.1 DO-178C, чтобы показать релевантность истории эксплуатации изделия, должно быть выполнено следующее [3]:
 - а. Должен быть задокументирован и признан достаточным объем времени, в течение которого ранее разработанное ПО находилось в эксплуатации.
 - б. Конфигурация ранее разработанного ПО и среды должна быть известна, релевант-

на и находиться под контролем.

- c. Если ранее разработанное ПО изменялось в течение истории эксплуатации, необходимо оценить релевантность истории эксплуатации для его обновленной версии и показать, что она сохраняется.
- d. Предполагаемое использование ранее разработанного ПО должно быть проанализировано, чтобы показать релевантность истории эксплуатации изделия, то есть продемонстрировать, что программное обеспечение будет использоваться тем же образом.
- e. Любые различия между средой истории эксплуатации и средой, в которой будет установлено ранее разработанное ПО, должны быть оценены, чтобы убедиться в применимости этой истории.
- f. Необходимо выполнить анализ, чтобы убедиться, что для программного обеспечения, которое не использовалось в реальной эксплуатации, например отключенного кода, не запрашивается зачет на основе истории эксплуатации.

2. *История эксплуатации должна быть достаточной.* Помимо демонстрации релевантности истории эксплуатации, необходимо показать, что объем истории эксплуатации достаточен для удовлетворения целей системной безопасности, включая уровень ПО. История эксплуатации также должна надлежащим образом закрывать те разрывы по DO-178C, которые она призвана восполнить.

3. *Проблемы, выявленные в эксплуатации, должны собираться и анализироваться.* Чтобы заявить историю эксплуатации, необходимо продемонстрировать, что проблемы, возникавшие в течение истории эксплуатации ранее разработанного ПО, известны, задокументированы и приемлемы с точки зрения безопасности. Для этого требуются свидетельства наличия адекватного процесса регистрации сообщений о проблемах. Как отмечалось ранее, выполнить это на практике может быть непросто.

В *Software Service History Handbook* Федерального управления гражданской авиации США выделяются четыре категории вопросов, которые следует задавать при предложении зачета на основе истории эксплуатации изделия [16].

Это действительно хорошие вопросы. Если на них можно дать удовлетворительные ответы, заявить историю эксплуатации, возможно, будет реально. Если нет, выстроить убедительное обоснование для сертифицирующих органов будет трудно. Ниже приведены категории этих вопросов:

- 45 вопросов, связанных с регистрацией сообщений о проблемах

- 11 вопросов по эксплуатации, то есть по сравнению сходства эксплуатации ранее разработанного ПО в прежнем домене с целевым доменом
- 12 вопросов по среде, то есть по оценке вычислительной среды, чтобы убедиться, что среда, в которой ранее разработанное ПО работало в период истории эксплуатации, похожа на предлагаемую среду
- 19 вопросов по времени, то есть по оценке продолжительности истории эксплуатации и частот ошибок на основе данных, доступных из истории эксплуатации изделия

Для удобства конкретные вопросы приведены в приложении D. За дополнительной информацией по истории эксплуатации изделия следует обращаться к DO-178C, раздел 12.3.4 [3], DO-248C, discussion paper (дискуссионный документ) No. 5 [17], и к *Software Service History Handbook* Федерального управления гражданской авиации США [16].

Глава 25. Обратная разработка

Сокращения

Сокращение	Расшифровка
CAST	Certification Authorities Software Team, группа сертифицирующих органов по ПО
COTS	Готовое коммерческое ПО (commercial off-the-shelf)
EASA	European Aviation Safety Agency, Европейское агентство по авиационной безопасности
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
LAL	Less abstract level, менее абстрактный уровень
TBY	Требование высокого уровня к ПО (high-level requirement)
THY	Требование низкого уровня к ПО (low-level requirement)
MAL	More abstract level, более абстрактный уровень
PDS	Ранее разработанное ПО (previously developed software)
RE	Обратная разработка (reverse engineering)
SQA	Гарантия качества ПО (software quality assurance)

В этой главе определяется, что такое обратная разработка, рассматриваются связанные с ней проблемы и даются рекомендации высокого уровня о том, как восстанавливать данные ЖЦ ПО, необходимые для выполнения целей DO-178C (КТ-178С). Эта глава тесно связана с главой 24, поскольку в DO-178C обратная разработка признается альтернативным методом создания данных ЖЦ ПО, особенно для ранее разработанного ПО. Как и другие темы в этой книге, данный вопрос рассматривается с акцентом на программное обеспечение, критичное для безопасности, в области гражданской авиации. Эти концепции могут применяться и в других областях, критичных для безопасности.

Следует отметить, что значительная часть этой главы посвящена восстановлению данных ЖЦ ПО, то есть требований и проекта ПО, по исходному коду, поскольку именно это обычно является наиболее распространенным вариантом применения обратной разработки в авиационной программной индустрии. Однако эти концепции можно применять и в других сценариях, например начиная с объектного кода или документируя отсутствующие системные требования.

25.1 Что такое обратная разработка?

Определения обратной разработки различаются. DO-178C определяет ее так: «Процесс разработки данных ПО более высокого уровня на основе существующих данных ПО. Примеры включают разработку исходного кода по объектному коду или исполняемому объектному коду, а также разработку требований высокого уровня к ПО (ТВУ) по требованиям низкого уровня к ПО (ТНУ)» [1].

Раздел 12.1.4.d документа DO-178C поясняет, что обратная разработка может использоваться как подход к модернизации базовой версии: «Обратная разработка может использоваться для повторного создания данных ЖЦ ПО, которые являются недостаточными или отсутствуют для выполнения целей данного документа [DO-178C]. Помимо получения программного продукта, для выполнения целей процесса верификации ПО может потребоваться выполнение дополнительных действий» [1].[*]

Роджер Прессман определяет обратную разработку для программного обеспечения так: «Процесс анализа программы с целью создания представления программы на более высоком уровне абстракции, чем исходный код. Обратная разработка представляет собой процесс восстановления проекта ПО» [2].

В документе CAST-18 группы Certification Authorities Software Team (CAST)[†] обратная разработка объясняется следующим образом:

Обратная разработка представляет собой подход к созданию данных ЖЦ ПО, которые изначально не существовали, не могут быть найдены, являются недостаточными или недоступны для выполнения применимых целей DO-178B/ED-12B. Однако это не просто создание соответствующих данных ЖЦ ПО, а процесс получения гарантии того, что эти данные корректны, функциональность ПО понята и задокументирована, а ПО функционирует так, как задумано и требуется системе. Он включает восстановление требований и проекта ПО, а также выполнение соответствующих действий по верификации на надлежащем уровне, чтобы обеспечить целостность ПО, наличие и корректность всех данных ЖЦ ПО и достижение надлежащего уровня гарантии разработки [3].[‡]

В исследовательском отчете Федерального управления гражданской авиации США (FAA) под названием *Reverse Engineering Software and Digital Systems* Джорджа Романски и соавторов говорится:

Обратная разработка – это класс процессов разработки, которые начинаются с детализированных представлений реализации и с применением различных методов создают более общие, менее детализированные представления. Цель состоит в том, чтобы получить более абстрактные представления, которые можно использовать для понимания структуры и

замысла более детализированных представлений и рассуждения о них [4].

25.2 Примеры обратной разработки

Существует множество примеров, где обратная разработка может применяться или уже применялась, в том числе следующие:

- Готовое коммерческое ПО (commercial off-the-shelf software, COTS), например операционные системы реального времени или библиотеки, поставляемые поставщиком.
- Программное обеспечение, изначально разработанное по другому стандарту, например военному или автомобильному, такое как контроллер двигателя, драйвер шины сети контроллеров (controller area network, CAN) или компонент системы управления полетом.
- Программное обеспечение с открытым исходным кодом, например библиотеки времени выполнения для открытого компилятора GNAT для языка Ada, операционная система Linux или гипервизор Xen.
- Существующее программное обеспечение, которое после многих лет сопровождения стало слишком хрупким, чтобы его можно было модернизировать или исправлять.

25.3 Проблемы, которые необходимо учитывать при обратной разработке

Сертифицирующие органы выявили типовые проблемы, связанные с обратной разработкой, в документе CAST-18. В CAST-18 основное внимание уделено обратной разработке по исходному коду. Эти же проблемы отмечались и в специальных проблемных документах Федерального управления гражданской авиации США по конкретным проектам, когда предлагалось использовать обратную разработку. Ниже эти типовые проблемы поясняются [3,5].[*]

Проблема 1: отсутствие четко определенного процесса. Процесс обратной разработки артефактов ПО должен быть организованным и четко определенным. Слишком часто обратную разработку используют как компенсацию плохих практик разработки, и при этом не следуют документированному процессу. Если обратная разработка применяется, то процессы, мероприятия, критерии перехода и стратегии выполнения целей DO-178C должны быть задокументированы в планах и стандартах.

Проблема 2: отсутствие обоснования того, как выполняются цели DO-178C. Сертифицирующие органы часто замечали, что когда компании предлагают обратную разработку как модель жизненного цикла, они недостаточно объясняют, каким образом будут выполняться цели DO-178C [3]. Как сказано в CAST-18: «Обратную разработку следует использовать с осторожностью и только в хорошо обоснованных случаях, то есть для проекта, который уже использовался в ряде применений и показал высокий уровень целостности. Использование

обратной разработки при разработке нового программного обеспечения сертифицирующие органы настоятельно не рекомендуют» [3].

Проблема 3: отсутствие доступа к экспертам и исходным разработчикам. Когда данные ЖЦ ПО отсутствуют, часто необходимо получить доступ к исходным разработчикам, чтобы понять ход их мысли. Исходный код бывает трудно расшифровать, особенно если в нем мало комментариев или их нет вовсе и если он написан с прицелом на оптимизацию производительности. Согласно CAST-18, наиболее успешными оказываются те проекты по обратной разработке, у которых есть доступ к исходной команде разработки, особенно когда требуется разъяснение неоднозначных или сложных мест [3].

Проблема 4: сложный или плохо документированный исходный код. Если не применяются строгие стандарты кодирования, исходный код многих продуктов готового коммерческого ПО трудно читать. Мне самой доводилось изучать исходный код нескольких готовых коммерческих операционных систем реального времени, и я по собственному опыту знаю, насколько трудным он может быть. Такой код часто насыщен сложными структурами данных и указателями, а в довершение ко всему почти не содержит комментариев. Проблемы с исходным кодом могут возникать потому, что его не разрабатывали для среды, критичной по безопасности, или потому, что это был быстрый прототип, который изначально не предполагалось использовать как конечный продукт. Плохо документированный исходный код затрудняет оценку предполагаемой функции кода и проверку того, что восстановленные требования и проект ПО являются достаточными. CAST-18 хорошо это резюмирует: «Глубокое понимание кода является обязательным условием успешной обратной разработки. Плохо документированный или сложный код не является хорошим кандидатом для обратной разработки» [3].

Проблема 5: трудности абстрагирования. При восстановлении проекта ПО и требований по исходному коду трудно достичь надлежащего уровня абстракции. Прессман поясняет: «В идеале уровень абстракции должен быть настолько высоким, насколько это возможно» [2]. Переходить от низких уровней абстракции, например кода, к более высоким уровням абстракции, например требованиям, действительно трудно. У неверно выбранного уровня абстракции есть несколько последствий, и здесь отмечены два. Во-первых, если работа выполнена неправильно, проект ПО и требования начинают слишком напоминать исходный код. Тесты, проводимые по таким требованиям, дают очень мало пользы, поскольку оценивают не замысел программного обеспечения, то есть *что* оно делает, а его реализацию, то есть как оно это делает. По сути такие тесты лишь доказывают, что код является кодом; они не доказывают, что код делает именно то, что должен делать. Во-вторых, при отсутствии должного уровня абстракции в исходном коде может присутствовать нежелательная функциональность, которая не видна на системном уровне. Когда между системными требова-

ниями и требованиями к ПО существует большой разрыв по степени детализации, трудно подтвердить полноту требований системного уровня.

Здесь уместно поднять тревогу по поводу *псевдокода*. В проектах с обратной разработкой требования низкого уровня к ПО (ТНУ) нередко оформляют в виде псевдокода, который выглядит почти в точности как сам код. Уже это плохо, но ситуация становится еще хуже, когда такие проекты пытаются использовать тесты по этому псевдокоду для выполнения целей DO-178C по анализу структурного покрытия. В таком случае анализ структурного покрытия становится практически бесполезным. Дополнительные сведения о псевдокоде см. в главах 7 и 8.

Проблема 6: трудности трассируемости. Трассируемость тесно связана с проблемой абстракции. Если уровни абстракции установлены неправильно, обычно возникают две потенциальные проблемы трассируемости. Обе являются симптомами того, что разработчики в действительности не понимают, что делает ПО. Во-первых, трассирование может выполняться грубой силой, то есть связи добавляются просто потому, что код или проект ПО должны трассироваться на что-либо. Связи добавляются на основании похожих слов, а не на основании прочного понимания ПО. Во-вторых, требования могут быть определены как *производные* (derived requirements), чтобы избежать необходимости трассирования. Обе эти проблемы трассируемости могут маскировать нежелательную функциональность, существующую в коде.

Проблема 7: проблемы взаимодействия с сертифицирующими органами. В проектах по обратной разработке процесс взаимодействия с сертифицирующими органами часто выполняется неудовлетворительно. Иногда обратная разработка не указывается в планах и не согласовывается с сертифицирующими органами [3]. Слишком много раз мне приходилось рассматривать комплект планов, в которых указано, что проект использует каскадную модель жизненного цикла, а при анализе фактических данных выяснялось, что на деле это обратная разработка, причем без всякого плана.

25.4 Рекомендации по обратной разработке

Чтобы заранее устранить отмеченные проблемы, предлагаются следующие рекомендации. Некоторые из них не обязательно являются уникальными именно для обратной разработки.

Рекомендация 1: оценивать и обосновывать уместность обратной разработки. Прежде чем запускать работы по обратной разработке, важно оценить уместность такого подхода. Зрелый, хорошо проверенный код с большим опытом эксплуатации, например примеры из раздела 25.2, может быть подходящим кандидатом для обратной разработки. Однако обратную разработку кода прототипа, использовавшегося для валидации требований, обосновать может быть трудно. Некоторые проекты пытаются взять свой код быстрого прототипа как

есть и написать под него проектные данные и требования. Такой подход не рекомендуется, поскольку зрелость и стабильность кода и его проекта остаются неопределенными. Как объяснялось в главе 6, код прототипа может использоваться как вход для требований, проектных данных и окончательного кода. Однако использование кода прототипа как входа в другие процессы и данные ЖЦ ПО следует ограничивать, поскольку у такого кода может не быть чистой, робастной или даже безопасной архитектуры. По моему опыту, обратная разработка по коду прототипа обычно обходится дороже и занимает больше времени, чем если бы код просто переписали.

Рекомендация 2: честно признавать, что используется обратная разработка. За последние 10 лет я рассмотрела много проектов, в которых применялась та или иная форма обратной разработки, однако большинство из них никогда не признавали этого в своих планах. Чтобы обратная разработка использовалась эффективно, она должна быть запланирована и реализована должным образом. Должно быть ясно, почему она применяется, как именно она будет выполняться и как с ее помощью будут достигаться цели DO-178C.

Рекомендация 3: выполнять полный анализ пробелов. Прежде чем окончательно принимать решение о работах по обратной разработке, важно провести полный анализ пробелов в существующих данных, чтобы определить, какие цели DO-178C выполняются, а какие нет. Часто проводится быстрая оценка, и график проекта строится по ее результатам. Но спустя несколько месяцев выясняется, что пробелов гораздо больше, чем было выявлено изначально. Чтобы избежать этого риска, следует собрать специальную команду квалифицированных инженеров для тщательного анализа и выявления пробелов как в данных, так и в процессах. В команду должны входить талантливые и опытные инженеры, обладающие техническим опытом по всему ЖЦ ПО, знанием предметной области и опытом работы с DO-178C.

Рекомендация 4: документировать жизненный цикл, процессы и критерии перехода. Обратная разработка сама по себе является жизненным циклом, и она должна быть документирована в планах так же, как и любой другой жизненный цикл. Следует определить фазы работ по разработке и верификации, включая входные данные, критерии входа, выполняемые мероприятия, выходные данные и критерии завершения для каждой фазы. Например, на рисунке 25.1 показан общий процесс для фазы обратной разработки.

На рисунке показан менее абстрактный уровень (less abstract level, LAL), который используется как вход для разработки более абстрактного уровня (more abstract level, MAL). Он показывает, что LAL рассматривается до разработки MAL, чтобы обеспечить качество LAL и его соответствие набору стандартов, например стандартам кодирования. После создания MAL он также рассматривается. Затем MAL и LAL рассматриваются вместе. Также следу-

ет отметить, что общий процесс показывает наличие запросов на изменение. Процесс не предполагает, что LAL идеален, тем самым избегая феномена код – король, то есть предположения, что код совершенен.

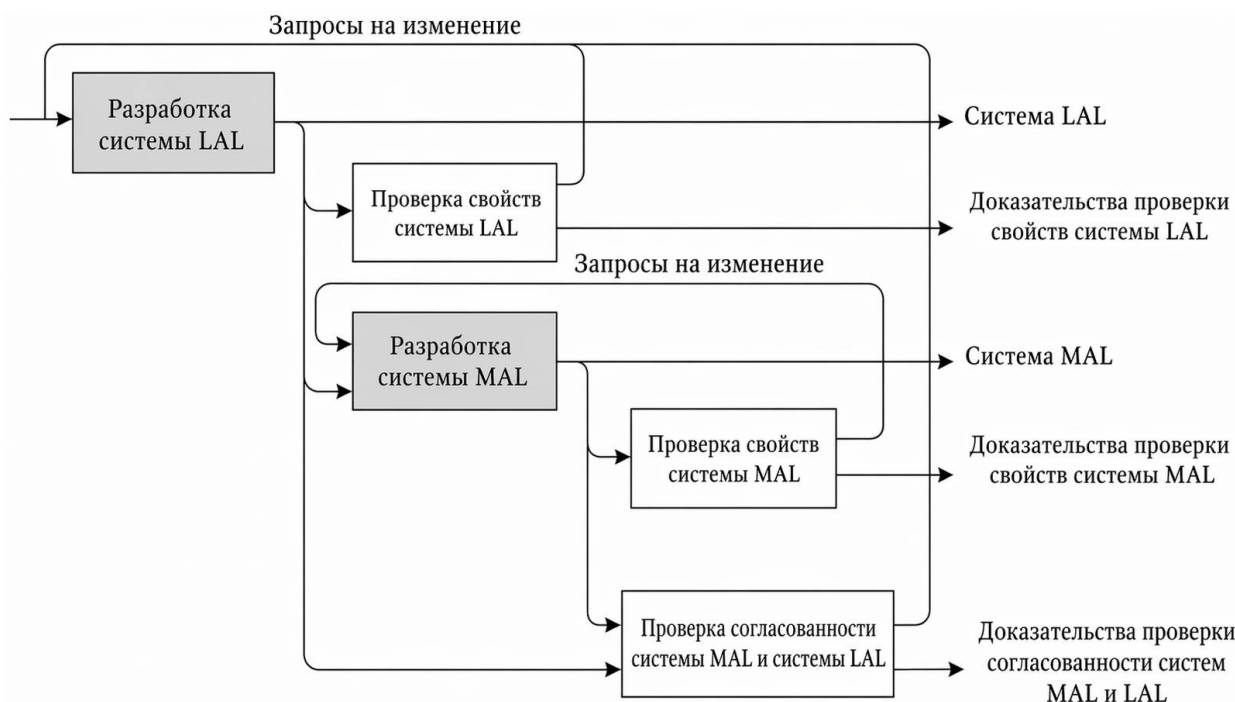


Рисунок 25.1 Общий жизненный цикл для различных уровней абстракции. (Из: G. Romanski et al., Reverse engineering software and digital systems, проект отчета к публикации FAA/DOT Office of Aviation Research, Washington, DC, October 2011. Используется с разрешения автора.)

Рекомендация 5: документировать детальные процедуры и стандарты. Реализация успешного и воспроизводимого процесса обратной разработки требует подробных процедур, контрольных перечней и стандартов. Например, если проект по обратной разработке начинается с исходного кода, исходные стандарты кодирования могут оказаться недостаточными или вообще отсутствовать. Поэтому потребуется разработать подходящие стандарты кодирования и рассмотреть код на соответствие этим стандартам. Также могут потребоваться специальные процедуры управления изменениями в коде. В частности, изменения нужно будет согласовывать со всеми соответствующими сторонами и утверждать до их реализации.

Рекомендация 6: согласовывать планы и добиваться согласия. Как и в любом проекте, который стремится получить одобрение сертифицирующих органов, планы и стандарты должны быть рассмотрены, внутренне утверждены, согласованы с сертифицирующим органом и утверждены им. Работы по обратной разработке могут потребовать дополнительного согласования с учетом ранее упомянутых сложностей. Это следует учитывать. Как отмечалось в главе 5, чем раньше планы будут согласованы и одобрены сертифицирующим органом, тем

лучше.

Рекомендация 7: координироваться с несколькими заинтересованными сторонами. В зависимости от характера проекта может существовать несколько заинтересованных сторон, включая системную команду, исходных разработчиков ПО, заказчика, а иногда и нескольких заказчиков, команду обратной разработки и так далее. Важно обеспечить, чтобы все заинтересованные стороны были определены, проинформированы и выполняли свои задачи так, как ожидается. Проекты с несколькими заинтересованными сторонами должны делать следующее [4]:

- Четко определять роли и ответственность.
- Определять и согласовывать используемые процессы.
- Описывать, согласовывать и верифицировать управление конфигурацией между заинтересованными сторонами.
- Согласовывать отслеживание проблем и неисправностей между всеми заинтересованными сторонами. Контролировать и отслеживать поток информации между заинтересованными сторонами.
- Обеспечивать наличие у заинтересованных сторон необходимой компетентности для выполнения своих обязанностей.
- Обеспечивать, чтобы коммуникация между заинтересованными сторонами не была затруднена.

Рекомендация 8: собрать все существующие данные и применить управление конфигурацией ПО, включая управление изменениями и регистрацию сообщений о проблемах. Существующие артефакты должны быть зафиксированы в базовой версии и переведены под управление изменениями до начала процесса обратной разработки. Например, исходный код, руководства пользователя, документы с требованиями или проектные данные, используемые как входы в работы по разработке, должны быть собраны, зафиксированы в базовой версии и взяты под контроль. Любые изменения должны проходить через процесс управления изменениями с использованием сообщений о проблемах и/или запросов на изменение.

Рекомендация 9: привлекать специалистов по гарантии качества ПО и по взаимодействию с сертифицирующими органами. Как и во всех проектах, важно привлекать персонал гарантии качества ПО и специалистов по взаимодействию с сертифицирующими органами. Для работ по обратной разработке это особенно важно. Поскольку обратная разработка считается решением с повышенным риском, сертифицирующие органы нередко выдают проектно-специфические указания, например проблемные документы Федерального управления гражданской авиации США или позиции по сертификационной проверке Ев-

ропейского агентства по авиационной безопасности (EASA). Раннее и постоянное участие персонала гарантии качества ПО и специалистов по взаимодействию с сертифицирующими органами помогает заранее устранять такие опасения.

Рекомендация 10: использовать технически сильную команду с предметной экспертизой. Обратная разработка – не работа для начинающих инженеров. Поскольку она связана с созданием более абстрактных данных на основе менее абстрактных, обратная разработка требует инженеров с технической экспертизой и знанием предметной области. Трудно, если вообще возможно, ожидать, что инженер по системам управления полетом успешно выполнит обратную разработку операционной системы реального времени. Точно так же даже в целом компетентный инженер, который не проходил полный жизненный цикл разработки на нескольких проектах, плохо подходит для такой работы. По моему опыту, успех проектов обратной разработки прямо пропорционален опыту инженеров, которые их выполняют. Для качественной обратной разработки нужна многопрофильная команда с сильными аналитическими способностями и хорошими навыками коммуникации, чтобы создавать данные ЖЦ ПО. Инструменты могут помогать, но успех проекта будет в значительной степени зависеть от качества инженеров.

Рекомендация 11: консультироваться с исходными разработчиками. Если это возможно, чрезвычайно полезно координироваться с исходными разработчиками ПО. Некоторые из них могут быть по-прежнему доступны, и их понимание может определить разницу между успехом и провалом. Они, возможно, не смогут играть активную роль в проекте, но даже небольшое количество содержательного времени, проведенного с ними по ходу проекта, приносит пользу. Такое взаимодействие существенно улучшает понимание ПО и качество артефактов. Поскольку возможностей консультироваться с исходными разработчиками может быть немного, это время нужно использовать разумно. Следует заранее подготовить перечень конкретных вопросов и убедиться, что ответы поняты полностью. Следует отметить, что в исследовании FAA по обратной разработке доступ к предметным экспертам, предпочтительно к исходным разработчикам, рассматривается как необходимость [4].

Рекомендация 12: стремиться к всестороннему пониманию функциональности ПО. При обратной разработке важно понимать функциональность и поведение ПО. По сути, обратную разработку можно рассматривать как процесс выявления поведения [4]. Критически важно учитывать общую картину, то есть что делает ПО, а не только детали низкого уровня.

Рекомендация 13: мыслить сверху вниз. Оценив десятки проектов, я обычно могу распознать работу, выполненную методом обратной разработки, даже если в планах указано иное, поскольку представление сверху вниз оказывается неполным. Иными словами, при постро-

нии нисходящих нитей требований, от системных требований к требованиям высокого уровня к ПО, затем к требованиям низкого уровня к ПО, оттуда к проекту ПО и далее к коду, обнаруживаются разрывы и недостающие детали реализации. Представление снизу вверх может выглядеть хорошо, например весь код трассируется на требования низкого уровня к ПО, а все требования низкого уровня к ПО трассируются на требования высокого уровня к ПО, но представление сверху вниз получается *грубым*. Основная проблема, которую я вижу, заключается в том, что связи трассируемости не показывают, что требования реализованы полностью. Даже если данные создаются снизу вверх, важно постоянно держать в уме представление сверху вниз. В исследовательском отчете FAA по обратной разработке объясняется необходимость «подчеркнуть, что движение снизу вверх не может гарантировать, что все желаемые функции на системном уровне будут присутствовать» [4]. Должен существовать некоторый «способ внесения реального системного замысла» и установления «полноты восстановленного представления о системном замысле» [4].

Рекомендация 14: оценивать робастность. При обратной разработке, особенно по коду, важно оценивать робастность ПО. Ранее разработанное ПО или код прототипа мог быть создан без расчета на робастную работу при неожиданных или аномальных входных данных. Любая недостающая функциональность, связанная с робастностью, должна быть зафиксирована в сообщении о проблеме и должным образом устранена.

Рекомендация 15: разрабатывать корректные уровни абстракции. Как уже обсуждалось, обратная разработка связана с разработкой более абстрактных уровней данных на основе менее абстрактных уровней данных, например проекта ПО по коду и требований высокого уровня по проекту ПО. Разрабатывать такие более абстрактные уровни трудно. Это сложно и в прямой разработке, а при обратной разработке еще сложнее. Переходить от большего количества деталей к меньшему непросто. «Следует тщательно учитывать различие между уровнями абстракции, чтобы обеспечить достаточную интеллектуальную добавленную ценность, демонстрирующую глубокое понимание двух представлений, которые трассируются и верифицируются» [4]. Именно здесь помогают опытные инженеры. Когда обратную разработку выполняет неопытный или недостаточно квалифицированный инженер, требования низкого уровня начинают выглядеть как сам код.

В исследовательском отчете FAA это сформулировано удачно:

Различие между уровнями абстракции должно обеспечивать баланс между величиной различия между каждым уровнем абстракции и количеством уровней абстракции. Более абстрактное представление должно описывать предполагаемое поведение менее абстрактного.

1. Простое повторное изложение предполагаемого поведения на том же или близком

уровне, но в другой нотации, полезно только для проверки того, что преобразование выполнено правильно, но не для проверки самого предполагаемого поведения.

2. Если различие между уровнями абстракции слишком велико, будет меньше уверенности в том, что процесс воспроизводим с теми же или эквивалентными результатами.

Эта проблема относится и к прямой разработке, однако у инженеров может возникать соблазн разрабатывать требования низкого уровня по коду так, чтобы они описывали то, как код работает, а не предполагаемое поведение. В результате различие между уровнем абстракции требований низкого уровня и требований высокого уровня может оказаться слишком большим [4].

Рекомендация 16: заранее обеспечивать трассируемость данных. Двухнаправленная трассируемость важна как для проектов прямой разработки, так и для проектов, где использована обратная разработка. В проектах обратной разработки она особенно важна, когда некоторые артефакты верхнего уровня, например системные требования или требования к ПО, уже существуют, а задача состоит в создании согласованных артефактов нижнего уровня, таких как проект ПО и код. То есть данные нижнего уровня восстанавливаются методом обратной разработки, чтобы соответствовать данным верхнего уровня. Двухнаправленная трассируемость поможет обеспечить согласованность как в представлении сверху вниз, так и в представлении снизу вверх.

Рекомендация 17: искать ошибки и планировать изменения кода. Одной из самых частых грубых ошибок при обратной разработке является отношение к коду как к царю или как к золотому эталону, то есть как к совершенному коду. Хороший процесс обратной разработки исходит из того, что у кода могут быть проблемы. В нем может быть ненужная функциональность, может не хватать робастности, комментарии могут быть плохими, а сам код может быть чрезмерно сложным. В нем могут быть даже логические или функциональные ошибки. Опрос, проведенный в рамках спонсируемого FAA исследования по обратной разработке, показал, что большинство проблем, поднятых в ходе работ по обратной разработке, относилось к исходному коду, 71% [4]. В отчете FAA также отмечалось, что большинство ошибок в исходном коде обнаруживается ручными методами: «Характерное наблюдение состоит в том, что наиболее продуктивным методом обнаружения ошибок при выполнении обратной разработки является ручной анализ, который выполняется как процесс разработки и приводит к созданию ТНУ [low-level requirements, требований низкого уровня к ПО] и установлению трассируемости между ТНУ [low-level requirements, требованиями низкого уровня к ПО] и исходным кодом» [4].[*] В исследовательском отчете FAA также предупреждалось, что «если выполняется обратная разработка рабочей кодовой базы, сам факт того, что кодовая база работает, может привести к ложному чувству уверенности в корректно-

сти программного обеспечения, что, в свою очередь, может привести к менее тщательному исследованию или меньшей аккуратности при разработке артефактов обратной разработки» [4]. Процесс обратной разработки должен искать ошибки, слабые места, сложности, неоднозначности и тому подобное в коде, которые могут повлиять на безопасность, а затем заранее устранять эти проблемы, например создавая безопасное подмножество кода.

Рекомендация 18: искать пробелы. При сведении вместе различных уровней абстракции важно искать и выявлять отсутствующие элементы. Иначе говоря, нужно искать ошибки пропуска, функциональность, которая должна присутствовать в коде и требованиях, но отсутствует. Также нужно искать несогласованности между различными уровнями абстракции, включая отсутствующие или ошибочные связи трассируемости, неполные системные требования или требования к ПО, ошибочный проект ПО и неиспользуемый код. Выявлять то, чего нет, труднее, чем рассматривать то, что есть. Это еще один случай, когда опытные инженеры способны дать ценное понимание: увидев множество проектов, они знают, чего ожидать и чего может не хватать.

Рекомендация 19: документировать проблемы по мере их выявления. При обратной разработке важно документировать потенциальные и фактические проблемы тогда, когда они замечены. Иначе о них можно забыть или упустить их из виду. Для этого есть несколько способов. Один из них состоит в ведении списка расследуемых вопросов, который периодически анализируется. Затем реальные проблемы переносятся в сообщения о проблемах. Другой подход состоит в открытии отдельного сообщения о проблеме по каждому вопросу. Если в итоге выяснится, что это не проблема, такую запись можно легко отменить или закрыть без действий. Как отмечалось в главах 8 и 10, важно указывать человека, который обнаружил проблему, а также инженера, который лучше всего разбирается в данной теме.

Рекомендация 20: устранять выявленные проблемы. Вопросы, классифицированные как реальные проблемы, должны быть устранены. Среди часто встречающихся проблем – неинициализированные переменные, проблемы с указателями, несогласованные определения данных, несогласованное использование данных, типов и единиц измерения, неверные алгоритмы, недостаточная интеграция модулей, незрелый проект запуска, как холодного, так и теплого, отсутствие обработки всех функциональных сценариев, неверный порядок событий, отсутствие встроенных тестов и передача данных, которые затем не используются [6].

Рекомендация 21: валидировать требования. DO-178C исходит из того, что системные требования, распределенные на ПО, валидированы. Поэтому DO-178C не требует валидации требований, то есть подтверждения того, что требования полны и корректны. Однако если системные требования восстановлены методом обратной разработки, они требуют валида-

ции, чтобы убедиться, что это именно те требования, которые нужны. Системные требования следует валидировать до проведения рассмотрения требований к ПО, проекта ПО и кода. Кроме того, любые требования к ПО, классифицированные как *производные* (derived requirements), то есть не трассирующиеся на системные требования, должны проходить валидацию.

Рекомендация 22: выполнять прямую верификацию. Чтобы выполнить цели DO-178C, любые данные разработки, полученные методом обратной разработки, необходимо верифицировать в прямом направлении. Иначе говоря, разработка снизу вверх допустима, однако нисходящая верификация все равно необходима. То есть нужно проводить рассмотрение требований высокого уровня относительно системных требований, рассмотрение требований низкого уровня относительно требований высокого уровня, рассмотрение исходного кода относительно требований низкого уровня и так далее. Прямая верификация также подтверждает нисходящую согласованность.

Рекомендация 23: после создания надежной базовой версии требований переходить к прямой разработке. Как только сформирована надежная базовая версия требований и имеются поддерживающие данные ЖЦ ПО, проект должен перейти к прямой разработке. Настоятельно не рекомендуется продолжать обратную разработку после того, как требования уже определены.

Рекомендация 24: понимать, когда следует остановиться. Не все проекты обратной разработки будут успешными. Если код в начале работ по обратной разработке не является зрелым или стабильным, это может привести к значительной переработке кода. Такая ситуация быстро становится дорогой и неуправляемой. В исследовательском отчете FAA говорится: Если продукт нестабилен и требуется много изменений в кодовой базе, затраты становятся очень высокими. Разработка требований к ПО, которое в этот момент обновляется, поглощает много ресурсов, поскольку работа повторяется, а из-за распространения влияния проблема становится больше и выходит из-под контроля. Если продукт нестабилен, процессы обратной разработки следует остановить и до их продолжения выполнить неформальные шаги по отладке [4].

Я бы пошла еще дальше и предположила, что в некоторых ситуациях код следует отбросить и начать заново. Иногда быстрее переписать чистый код, чем пытаться спасти сломанный или хрупкий.

Рекомендация 25: учитывать, что обратная разработка – это трудно. Полноценная обратная разработка от начала до конца не является легкой задачей. Она требует значительных усилий и должной тщательности. В некотором смысле она может быть даже сложнее прямой разработки, потому что подниматься вверх по уровням абстракции может быть труднее,

чем декомпозировать вниз.

Рекомендация 26: документировать извлеченные уроки. Как и в любом жизненном опыте, важно документировать извлеченные уроки, чтобы не усваивать их заново. На протяжении проекта рекомендуется вести перечень извлеченных уроков и регулярно оценивать, что работает, а что нет.

Литература

1. RTCA DO-178C, *Software Considerations in Airborne Systems and Equipment Certification* (Washington, DC: RTCA, Inc., December 2011).
2. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 4 edn. (New York: McGraw-Hill, 1997).
3. Certification Authorities Software Team (CAST), Reverse engineering in certification projects, Position Paper CAST-18 (June 2003, Rev. 1).
4. G. Romanski, M. DeWalt, D. Daniels, and M. Bryan, Reverse engineering software and digital systems, Draft report to be published by FAA/DOT Office of Aviation Research (Washington, DC, October 2011).
5. L. Rierson and B. Lingberg, Reverse engineering of software life cycle data in certification projects, IEEE Digital Avionics Systems Conference (Indianapolis, IN, 2003).
6. C. Dorsey, Reverse engineering within a DO-178B framework, Federal Aviation Administration National Software Conference (Danvers, MA, 2001).

*Скобки добавлены для пояснения.

†CAST – это группа международных сертифицирующих органов, стремящаяся гармонизировать свои позиции по бортовому программному обеспечению и авиационной электронной аппаратуре в документах CAST.

‡ Документ CAST-18 был написан до публикации DO-178C и поэтому ссылается на DO-178B.

*Приложение С исследовательского отчета FAA по обратной разработке содержит возможные меры снижения наиболее распространенных проблем [4].

*Скобки добавлены для ясности.

Глава 26. Аутсорсинг и офшоринг мероприятий ЖЦ ПО

Сокращения

Сокращение	Расшифровка
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
PR	Сообщение о проблеме (problem report)
PSAC	План сертификации ПО (Plan for Software Aspects of Certification)
SAS	Итоговый отчёт разработки ПО (Software Accomplishment Summary)
SCI	Указатель конфигурации ПО (Software Configuration Index)
SOW	Описание объема работ (statement of work)
TSO	Технический стандартный приказ (Technical Standard Order)
U.S.	United States (Соединенные Штаты)

26.1 Введение

«Сорсинг означает передачу работы, обязанностей и прав на принятие решений кому-то другому» [1]. Аутсорсинг представляет собой передачу работ внешней организации, то есть внешнему субъекту. Аутсорсинг разработки и верификации программного обеспечения не нов, но сегодня он, безусловно, распространен сильнее, чем в прошлом. При аутсорсинге нужно учитывать множество аспектов, однако рассмотреть их все здесь невозможно. В этой главе внимание сосредоточено на передаче одной или нескольких командам части или всех работ по разработке и верификации критичного по безопасности ПО. В главе рассматриваются причины популярности аутсорсинга, потенциальные риски и трудности аутсорсинга и офшоринга, а также рекомендации по снижению этих рисков.

Эта глава рассматривает тему аутсорсинга с точки зрения безопасности и сертификации. Она в первую очередь написана для технических руководителей по программному обеспечению, которые принимают решения об аутсорсинге и управляют внешними ресурсами в проектах критичного по безопасности ПО; однако она также применима к разработчикам, верификаторам, специалистам по гарантии качества и персоналу, занимающемуся сертификацией. Здесь обсуждается много организационных аспектов программы, поскольку они влияют на жизненный цикл ПО и конечный продукт. Политические, юридические и договорные аспекты аутсорсинга не рассматриваются. Обратите внимание, что в этой главе аутсорсинг рассматривается с позиции Соединенных Штатов (United States, U.S.). Описанные идеи должны быть применимы и в других странах; однако для компаний из США при

офшорной разработке продукции гражданской авиации существуют некоторые специфические вопросы.

По моему опыту в авионике, аутсорсинг обычно попадает в одну из трех категорий: (1) изделия, например линейно заменяемые блоки или целые системы, (2) приложения ПО или электронной аппаратуры, либо (3) трудовые услуги, например части работ по разработке и/или верификации ПО. С точки зрения США существуют три географических варианта, где выполняется аутсорсинг: (1) внутри США, (2) вне США, в странах с двусторонними соглашениями с Федеральным управлением гражданской авиации США (Federal Aviation Administration, FAA), и (3) вне США, в странах без двусторонних соглашений с этим управлением. Местоположение важно по нескольким причинам, включая опыт, предметную экспертизу, правовую рамку и сертификационные требования. В частности, Федеральное управление гражданской авиации США учитывает местоположение организации, которой переданы работы, поскольку это напрямую влияет на то, кто отвечает за надзор в рамках сертификации. Проекты, выполняемые в США, контролируются этим управлением. Для работ, выполняемых в странах с двусторонними соглашениями с ним, управление может запросить выполнение надзора у соответствующего международного сертифицирующего органа.[*] В странах без двусторонних соглашений с ним отсутствует местный сертифицирующий орган, который управление могло бы юридически утвердить или признать; следовательно, это может потребовать от него дополнительных ресурсов для надзора за такими работами. Это может рассматриваться как *чрезмерная нагрузка* для правительства США, поскольку управление не является организацией, работающей по модели оплаты за услугу. Если аутсорсинговые работы классифицируются как *чрезмерная нагрузка*, полученные результаты могут не быть приняты этим управлением.

Опыт показывает, что программные работы, выполняемые офшорно, требуют особенно внимательного подхода. Поэтому сертифицирующие органы считают офшорные проекты более рискованными. Федеральное управление гражданской авиации США не различает аутсорсинговые, офшорные и субподрядные работы, все организации, выполняющие такие работы, называются просто *поставщиками*. Даже компаниям, владеющим удаленными площадками внутри или вне США, необходимо учитывать вопросы сорсинга.

В остальной части этой главы термин *аутсорсинг* включает субподрядные работы или работы дочерних структур, выполняемые внутри страны (onshore), в соседней стране (near-shore) или за рубежом (offshore). Эти термины не зависят от того, кому принадлежит организация, выполняющая работу. Некоторые яростно утверждают, что работа, выполняемая принадлежащей компании дочерней структурой, не считается *аутсорсингом*. Я согласна, что бизнес-схема в этом случае иная и риск должен быть ниже; однако многие проблемы

оказываются сходными. Поэтому я включаю такие случаи в категорию *аутсорсинга*. С точки зрения сертификации бизнес-схема менее важна, чем качество работы. История показывает, что многие офшорные проекты имеют проблемы с качеством, некоторые из которых описаны далее. По мере того как компании учатся лучше справляться с проблемами качества, это восприятие может измениться.

26.2 Причины аутсорсинга

Причин для аутсорсинга много; часть из них экономические, часть технические. Ниже приведен перечень распространенных причин:

- *Это помогает компенсировать нехватку инженерных кадров.* В западном мире, например в США, Канаде и Европе, ощущается нехватка инженеров-программистов. Чтобы удовлетворить потребности проектов, привлекают внешних инженеров, которые усиливают существующую команду компании и помогают выстраивать последовательность выполнения проектов. Это позволяет компании брать на себя дополнительную работу и направлять внутренние ресурсы и персонал на наиболее критичные проекты.
- *В некоторых странах стоимость труда ниже.* Более низкая стоимость труда также привлекательна для бизнес-руководителей. Однако, по моему опыту, стоимость обычно пропорциональна качеству получаемой работы. Мой отец очень любит повторять: «Получаешь именно то, за что платишь». Из-за неопытности, неэффективности и слабого надзора мне доводилось видеть, как *дешевый* инженер в итоге обходился более чем в три раза дороже, чем опытный инженер с *высокой ставкой*. Рассматривая ставки оплаты труда, бизнес-руководители часто забывают, что внешние команды требуют надзора; соотношение

между числом инженеров внутри страны и числом офшорных инженеров для надзора колеблется от 1:1 до 1:30 в зависимости от характера передаваемой работы и конкретной команды. Когда офшорная команда становится более опытной, ее ставки обычно растут, и она перестает быть *дешевой*. Руководителей, которые думают, что сэкономят целое состояние за счет низких ставок внешних исполнителей, стоит заранее предупредить: в разработке критичного по безопасности ПО бесплатных обедов не бывает.

- *Некоторые компании обладают специализированными навыками.* Вместо того чтобы нанимать людей или развивать экспертизу в новой либо непрофильной области, например в операционных системах, может быть практичнее передать работу организации, которая на этой области специализируется. Это дает компании доступ к экспертизе, знаниям и возможностям, выходящим за пределы ее собственных ресурсов [1].
- *Это позволяет организовать круглосуточную работу.* Передача работы командам, на-

ходящимся в сильно отличающихся часовых поясах, может дать возможность круглосуточной продуктивности. В этом есть и плюсы, и минусы. Круглосуточный график означает множество поздних ночных и ранних утренних совещаний или телеконференций, что способно привести команду к выгоранию. Поддержка круглосуточной работы также требует глобальной инфраструктуры, которую нужно создать еще до запуска проекта.

- *Это предписано.* Некоторый аутсорсинг, особенно в офшорные страны, навязывается высшим руководством по иным бизнес-соображениям или условиями авиационных контрактов. Обычно такие предписания сопровождаются установленной долей или объемом работ, который должен быть передан определенным странам или организациям. К сожалению, именно такие проекты часто испытывают наибольшие трудности при попытке эффективно организовать аутсорсинг.

Если все сделано правильно, аутсорсинг может оказаться выгодным для всех участников. Однако я видела достаточно провалов аутсорсинга, как внутри США, так и за их пределами, чтобы понимать: для успеха требуется чрезвычайно бережный подход и постоянный надзор.

Для аутсорсинга не существует ни волшебной формулы, ни универсального решения на все случаи. В оставшейся части главы приведены предостережения и предложения о том, как подходить к этим вопросам.

26.3 Трудности и риски аутсорсинга

Практики аутсорсинга сильно различаются, от нескольких задач, переданных отечественному поставщику, до целой системы, переданной офшорной организации в стране без двусторонних соглашений, а между этими крайностями существует множество вариантов. В этом разделе рассматриваются некоторые типовые трудности. Рекомендации по снижению этих трудностей обсуждаются в следующем разделе. Первые девять трудностей относятся и к внутренним, и к офшорным командам. Последние восемь пунктов больше относятся к офшорным работам.

Обратите внимание, что далее в этом разделе внешняя команда называется *поставщиком* независимо от вида выполняемой работы или характера деловых отношений. Поставщик – это организация, выбранная для выполнения части или всех работ по проекту критичного по безопасности ПО. Кроме того, компания, использующая аутсорсинговую команду, то есть поставщика, далее в этом разделе называется *заказчиком*. Предполагается, что именно заказчик в конечном счете несет ответственность за соответствие DO-178C (KT-178C).

Трудность 1: Недостаток опыта. Если у поставщика нет достаточного опыта в программ-

ной инженерии, управлении, сертификации и вопросах безопасности, успех маловероятен. Много лет назад, когда мне было под сорок, я посетила офшорного поставщика по проекту уровня А. Я чувствовала себя бабушкой. Почти вся команда только что закончила колледж. Я уважаю молодость, но она должна быть уравновешена опытом и направляться им. Есть известное выражение: «Если хочешь победить, не выставляй мулов на Кентуккское дерби». Чтобы выиграть Кентуккское дерби, нужна чистокровная скаковая лошадь. Точно так же для разработки и верификации высококачественного программного обеспечения нужны талантливые и опытные инженеры.

Трудность 2: Недостаток знания предметной области. Если инженеру не хватает опыта в предметной области, качество работы страдает. Инженер может быть очень способным, но у него может не быть знаний, необходимых для принятия обоснованных решений. Передавать разработку и верификацию встроенного программного обеспечения, критичного по безопасности, компании с опытом в телекоммуникациях, вероятно, не лучшая идея для большинства авиационных программных проектов, поскольку таким инженерам обычно не хватает понимания ПО реального времени, детерминированного ПО и ПО, определяемого требованиями безопасности, а также дисциплины и строгости, которых требует DO-178C. Недавно я консультировала проект уровня А, в котором руководитель по программному обеспечению впервые занимался авионикой и сертификационным проектом. До этого он работал с оборудованием для сельского хозяйства. Это был способный инженер; однако он не понимал ни систем управления полетом, ни вопросов безопасности, ни целей DO-178B, ни ожиданий сертификации. Несмотря на то, что иногда думают иначе, авионика требует опыта в предметной области, приверженности высокому качеству и перфекционистского склада ума. Если нужен качественный результат, инженеров нельзя просто переставлять как фишки на игровой доске.

Трудность 3: Нестабильность кадрового состава. Для некоторых поставщиков проблемой может быть высокая текучесть персонала. Текучесть может быть вызвана низким моральным состоянием, недостаточной оплатой, отсутствием возможностей для поощрения, динамикой роста страны и так далее. На создание сильной инженерной команды с правильными людьми на правильных позициях уходят годы. Если состав постоянно меняется, приходится все время переобучать людей и снова начинать с нижней части, или почти с нижней части, кривой обучения. Это приводит к снижению качества работы.

Трудность 4: Нереалистичные ожидания. Многие работы проваливаются из-за нереалистичных ожиданий. Во многих проектах используется то, что я называю *планированием, основанным на успехе*. Это происходит, когда руководители проекта не закладывают в план кривые обучения, время на передачу работ, вероятные темпы выполнения, неудачные тесты,

замечания по аудитам или зависимости в графике, которые неизбежно возникают. В результате проекты изначально подготавливаются к провалу. Нереалистично ожидать от непроверенной команды программного обеспечения высокого качества при сжатом графике. Иными словами, ожидать слишком многого и слишком быстро просто неразумно. Давление сроков часто ведет к плохим решениям, например к попытке забросать проблему большим числом инженеров, вместо того чтобы дать правильным и квалифицированным инженерам спокойно сделать свою работу как следует. Точно так же нереалистично ожидать, что неопытная команда будет выпускать качественные продукты. Крутые кривые обучения не всегда вообще удается преодолеть. Для хорошего качества нужны достаточные ресурсы, разумный график и опытный персонал.

Трудность 5: Быстрый рост. Успех может привести к катастрофе. Некоторые поставщики растут слишком быстро. Стремясь удовлетворить потребности нескольких заказчиков или проектов, они могут чрезмерно расплываться, подтягивать для помощи менее опытных людей или перебрасывать опытных инженеров между проектами. Сложно обеспечить, чтобы нужные люди были в нужном месте в нужное время. Одна компания, которую я консультирую, предлагает удваивать или утраивать численность персонала каждый год в течение следующих пяти лет в стране, где почти нет опыта создания критичного по безопасности программного обеспечения. На мой взгляд, это не план, основанный на реальности.

Трудность 6: Приукрашивание возможностей компании и ее компетенций. У некоторых поставщиков программного обеспечения есть очень убедительные маркетологи, но при этом они не способны реально выполнить обещанное. Я присутствовала на нескольких маркетинговых презентациях, где представитель по продажам жонглировал DO-178B или DO-178C и сертификационным жаргоном так, будто прекрасно разбирается в теме. Но стоило начать задавать вопросы, как становилось очевидно, что это выступление мало отличается от телевизионной рекламы. Покупателю стоит быть осторожным. Обманчивый маркетинг бывает не только на рынке поддержанных автомобилей. Даже добросовестные поставщики иногда переоценивают собственные возможности.

Трудность 7: Изменяющиеся процессы. Некоторые компании хорошо работают, пока следуют собственному процессу. Однако когда процесс меняется, например из-за потребностей заказчика или особенностей предметной области, они могут адаптироваться плохо. По моему опыту, чем больше команда, тем труднее изменить процесс. Трудно обучить всех и добиться соблюдения изменений, особенно если участники команды не понимают, зачем эти изменения нужны.

Трудность 8: Плохой процесс выбора. Когда решается, какая работа и где будет выполняться, важно дать объективную оценку. Очень часто команды вынуждены выбирать опреде-

ленных поставщиков, например по местоположению или из-за маркетинговых связей, а не лучшего поставщика. Неспособность правильно оценить навыки, глубину компетенций и послужной список поставщика приводит к проблемам в дальнейшем. Однажды я наблюдала проект, где на работу претендовали три возможных поставщика. Выбрали того, кто дал самую низкую оценку стоимости, хотя были явные признаки того, что этот претендент не сможет выполнить работу качественно и в срок. В итоге это предприятие обошлось как минимум в десять раз дороже первоначального предложения самого дешевого участника, что соответствовало примерно тройной стоимости предложения более квалифицированного поставщика. И, к слову, проект опоздал на два года.

Трудность 9: Низкое качество. При работе с аутсорсинговыми поставщиками часто трудно добиться качественного результата. Такая проблема возможна и внутри одной компании, но в другой организации ее обычно сложнее выявить до более поздней стадии процесса или жизненного цикла, когда менять направление, возможно, уже поздно. Качество – это не просто отслеживание метрик. Без надлежащего технического надзора можно только под конец проекта обнаружить, что требования плохие, проект ПО ошибочен, а код не работает. Иногда даже опытные компании дают сбой по качеству: возможно, они слишком растянулись, наняли нового руководителя программы, который еще не готов к этой роли, или поддались давлению сроков.

Трудность 10: Языковые проблемы. Взаимодействие с некоторыми офшорными поставщиками может быть осложнено языком. Если команда не владеет английским достаточно уверенно, [*] трудно писать недвусмысленные требования, точно проверять требования, составлять полезные сообщения о проблемах, обсуждать вопросы и так далее. Мне доводилось консультировать проекты, где для общения требовался переводчик. Это может занимать очень много времени и приводить к ошибкам. Если языковые проблемы сочетаются с недостатком опыта и/или отсутствием культуры безопасности, вероятность успеха заметно падает. Раз уж речь зашла о языке, следует отметить, что Федеральное управление гражданской авиации США требует, чтобы данные, относящиеся к сертификации, были на английском языке и были доступны в США [2].

Трудность 11: Культурные различия. При аутсорсинге важно понимать культурные различия. Культурные различия бывают и внутри США, но наиболее серьезные трудности возникают в международных проектах. Вот несколько примеров культурных аспектов, о которых следует помнить:

- Разные культуры часто по-разному мотивируются. То, что работает для мотивации команды в США, может не работать в Европе, Азии или Южной Америке.
- В некоторых странах считается неуважительным указывать на негативные стороны в

человеке или в чем-либо еще. Однако при верификации ПО цель состоит в том, чтобы выявлять ошибки. Без правильного отношения к верификации можно получить самую успешную программу тестирования в истории, а система все равно не будет работать.[*]

- Не во всех странах одинаковы стандарты безопасности или ценность человеческой жизни. Оценка дефектов программного обеспечения требует установки на нулевую дефектность для критичного по безопасности ПО. В одном случае офшорная компания, которую я консультировала, не понимала, зачем вообще тестировать или исправлять ошибки в ПО уровня А.
- В некоторых культурах могут не уважать совершенство или вообще не иметь такого понятия. В ряде культур даже может считаться неправильным гордиться своей работой. Однако разработка критичного по безопасности программного обеспечения требует перфекционистского подхода и внимания к деталям. Если инженеров устраивает посредственная или просто *достаточно хорошая* работа, доказать безопасность будет трудно.

У каждой культуры есть свои сильные и слабые стороны. Важно понимать культурный фон всех участников, проводить культурную подготовку для всех вовлеченных членов команды и закладывать время на интеграцию многообразия.

Трудность 12: Этические проблемы. Не все культуры придерживаются одинаковых этических стандартов. В некоторых культурах считается допустимым лгать ради продвижения вперед, воровать, если тебя не поймали, и предлагать взятки. Некоторые страны подозреваются во внесении уязвимостей безопасности и скрытых лазеек в разрабатываемую ими продукцию.

Трудность 13: Проблемы с образованием. Не все инженерные образовательные программы одинаковы по качеству. В некоторых учебных заведениях мира проходным считается результат в 50 процентов. Сам по себе диплом инженера еще не означает, что человек действительно умеет заниматься инженерной работой. Поэтому необходимо проверять качество образования инженеров поставщика, так же как и их опыт.

Трудность 14: Скрытые расходы. При офшоринге могут возникать, и скорее всего будут возникать, непредвиденные расходы. Авиабилеты бизнес-класса и безопасные гостиницы очень быстро складываются в заметные суммы. Несмотря на развитие видеоконференций и сетевых совещаний, личные встречи и очный надзор по-прежнему необходимы. Кроме того, создание инфраструктуры, например оборудования, высокоскоростного сетевого доступа и прочего, может обходиться дорого.

Трудность 15: Потеря опытных инженеров. Несмотря на все усилия представить аутсорсинг в положительном свете и поощрять командную работу, иногда при переходе к аутсорсингу компания теряет опытных инженеров. Инженеры заказчика могут воспринимать такую схему как передачу их работы на сторону, либо им может быть интереснее самим проектировать и писать код, чем надзирать за поставщиком. Дополнительным фактором может стать и выгорание из-за ранних утренних и поздних ночных совещаний.

Трудность 16: Проблемы сертификации. Как уже отмечалось ранее, сертифицирующие органы сталкивались с проблемами при аутсорсинге, особенно при офшоринге, поэтому обычно требуют специального обоснования и надзора за офшорными поставщиками. Если работа выполняется в стране без двусторонних соглашений, необходимо обоснование того, что это не создает чрезмерной нагрузки. Некоторые предложения о том, как этого добиться, рассматриваются в следующем разделе.

Трудность 17: Послепроектная амнезия. Мне доводилось консультировать проекты, которые я считала сертификационными провалами. Однако сам факт того, что они все же дошли до конца и не обанкротили компанию, каким-то образом начинает подаваться как успех. Я видела, как плохо проявившие себя аутсорсинговые компании получали новую работу лишь потому, что руководство словно забывало тот хаос, который происходил раньше. Маркетинговый отдел поставщика может даже объявить, что именно они были ключом к успеху проекта, разумеется, не упоминая те проблемы, которые сами же и создали. Если не предпринять конкретных действий для исправления ошибок прошлого, эти ошибки почти наверняка повторятся. Как я слышала не раз: «Безумие – это делать одно и то же снова и ожидать иного результата». К сожалению, когда речь идет об аутсорсинге, особенно офшорном, я видела достаточно такого безумия, чтобы стать скептиком.

26.4 Рекомендации по преодолению трудностей и рисков

Несмотря на трудности, описанные в предыдущем разделе, аутсорсинг может быть успешным. При правильной организации он способен дать значительные преимущества. В этом разделе представлены стратегии и рекомендации по работе с трудностями аутсорсинга. Как уже отмечалось, основное внимание здесь уделено вопросам безопасности и сертификации. Организованный процесс необходим для разработки программного обеспечения, которое выполняет свою предполагаемую функцию. Поэтому здесь затрагиваются и некоторые практические организационные меры, влияющие на общее качество ПО при аутсорсинге.

Еще в юности, в сельской Америке, мне хотелось увидеть мир. Инженерная работа открыла мне двери в этот мир. Мне нравится встречаться, работать и обедать с людьми со всего света. Взаимодействие с такими разными, интересными и талантливыми людьми по всему миру действительно расширяет взгляд. Нужно время, чтобы вырастить проект или команду

до состояния, в котором начинают проявляться преимущества аутсорсинга, но когда это удастся, результат может приносить большое удовлетворение. Эти рекомендации призваны помочь вам и вашей команде испытать именно такое удовлетворение.

Как и в предыдущем разделе, в этом разделе внешняя команда называется *поставщиком*, а компания, использующая аутсорсинговую команду, называется *заказчиком*.

Рекомендация 1: Определите готовность к аутсорсингу. Некоторые компании прибегают к аутсорсингу просто потому, что видят, как это делают все вокруг. Без должной подготовки и предварительной проработки аутсорсинг может обернуться весьма печальным провалом. Прежде чем входить в аутсорсинговую схему, важно оценить готовность заказчика к аутсорсингу. В своей книге *Software Without Borders* Стив Мезак приводит тест готовности к аутсорсингу из 20 вопросов для оценки готовности организации к такому шагу. Этот тест рассматривает опыт аутсорсинга, технологическую ситуацию, состояние бизнеса и управленческий подход [3].[*] Такая оценка может помочь понять, является ли аутсорсинг разумным решением и где может потребоваться дополнительная помощь.

Рекомендация 2: Обеспечьте приверженность высшего руководства. Крайне важно иметь как краткосрочную, так и долгосрочную поддержку со стороны руководства высшего уровня. Без этого провал могут объявить уже при первых признаках трудностей. Руководству нужно дать полное и точное представление о потребностях, рисках, вариантах и ожиданиях. И команда заказчика, и команда поставщика должны быть осторожны и не обещать того, что не смогут выполнить. Оптимизм допустим, но он должен быть уравновешен точным отражением реальности. Хорошо информированное и действительно вовлеченное руководство может открывать двери, обеспечивать поддержку и задавать видение как проекту, так и предприятию в целом.

Рекомендация 3: Используйте системный и организованный подход. Некоторые организации врываются в аутсорсинг обеими ногами еще до того, как у них появляется хоть какой-то план. Это тот самый подход *сначала стреляй, потом целйся* в разработке программного обеспечения. Никто не строит дом без чертежа, и точно так же нельзя эффективно передать на сторону часть или даже всю работу компании без организованного подхода. В книге *The Outsourcing Handbook* Марка Пауэрса и соавторов объясняется необходимость процессного подхода к аутсорсингу и предлагается жизненный цикл аутсорсинга из семи фаз.

Эти семь фаз цикличны, если только не реализуется вариант выхода, и каждая фаза выполняется независимо от следующей. Важно качественно проработать каждую из фаз и не пытаться проскочить ни одну из них. Ниже кратко рассматривается каждая фаза [1]:

1. *Стратегическая оценка.* Она включает построение бизнес-обоснования для выявления выгод аутсорсинговой схемы, включая анализ ключевых компетенций и областей,

пригодных для передачи на сторону.

2. *Анализ потребностей.* Он выполняется применительно к конкретному проекту для выявления его специфических потребностей.
3. *Оценка поставщика.* Она включает оценку поставщиков с целью определить, какой из них лучше всего соответствует конкретным потребностям. Сюда входят не только оценки управления программой, но и детальное техническое рассмотрение инженерной компетентности, глубины экспертизы, инфраструктуры и качества. «Выбор правильного поставщика во многом похож на выбор хорошего партнера; есть все шансы, что, приняв правильное решение с самого начала, вы получите потенциально долгосрочные отношения, тогда как выбор неправильного поставщика может повредить и сорвать аутсорсинговый проект, задуманный с благими намерениями» [1].*
4. *Управление контрактом и переговорами.* Это процесс согласования аутсорсингового соглашения и ввода контракта в действие. Для каждого поставляемого результата и контрольной точки должны быть установлены критерии качества и перехода. Мне доводилось видеть проекты, которые оплачивали поставленные результаты, а потом обнаруживали, что они непригодны и требуют полной переделки.
5. *Инициация проекта и переходная фаза.* Это этап, на котором работа передается поставщику. Именно эта фаза задает тон всему остальному проекту, поэтому важно быстро реагировать на возникающие вопросы. Для технической работы это особенно критичный этап. Проекты должны заранее предусматривать надлежащий уровень технического надзора, чтобы обеспечивать коммуникацию, согласование проблем и приемлемость поставляемых результатов.
6. *Управление отношениями.* Когда проект входит в более рутинный режим, важно поддерживать здоровые отношения между заказчиком и поставщиком. Эта фаза включает оценку отношений, решение проблем, управление коммуникацией, обмен знаниями и управление процессом.
7. *Стратегия продолжения, изменения или выхода.* В конце каждого проекта необходимо оценить аутсорсинговые отношения и определить, следует ли их продолжать, изменять или прекращать. Если принято решение продолжать, нужно проанализировать извлеченные уроки и внедрить улучшения. Если принято решение изменить процесс, требуется ясная стратегия изменений. Если принято решение выйти из схемы, для этого тоже нужна хорошо продуманная стратегия.

Рекомендация 4: Будьте хорошо информированы. Убедитесь, что для реализации схемы аутсорсинга доступны компетентные люди. Если аутсорсинг для вас в новинку, разумно

привлечь помощь. Возможно, сможет помочь стратег по аутсорсингу или другое подразделение предприятия. Нужно учитывать множество деталей, например юридически обязательные контракты, законы об импорте и экспорте, защиту интеллектуальной собственности, обучение культурной чувствительности [для офшорных проектов], оформление виз, передачу оборудования, сетевую инфраструктуру. Чрезвычайно полезно при первом таком опыте опереться на экспертов и при этом иметь выделенную команду, которая сможет максимально перенять их знания, чтобы заложить надежную основу.

Рекомендация 5: Знайте собственную организацию. Прежде чем запускать аутсорсинговую схему, важно понимать характеристики собственной организации. Определите сильные и слабые стороны, конкретные потребности, зрелость организации, техническую экспертизу, зрелость процессов и так далее. С технической точки зрения важно проанализировать ключевые и непрофильные компетенции. «Ключевые компетенции – это сочетания специальных навыков, собственных технологий, знаний, информации и уникальных операционных процессов и процедур, которые встроены в продукты и услуги организации и выступают уникальными отличиями для ее заказчиков» [1]. Непрофильные компетенции – это такие компетенции, которые не отличают организацию от конкурентов и не влияют напрямую на ее продукты и услуги. Они необходимы для повседневной работы и косвенно влияют на услуги и продукты организации [1]. Большинство компаний сначала передают на сторону именно непрофильные компетенции.

Рекомендация 6: Определите, что отдавать на сторону, а что оставлять внутри компании. Оценка ключевых компетенций помогает в решении этой задачи; однако после определения компетенций и потребностей нужна стратегия их покрытия. Ниже приведены некоторые распространенные подходы к построению проекта под аутсорсинг:

- Передать на сторону весь продукт целиком, включая системную часть, аппаратуру и программное обеспечение. Передать на сторону весь жизненный цикл ПО, а разработку и тестирование аппаратуры и систем оставить внутри компании.
- Разбить функциональность на модули так, чтобы одни функции выполнялись внутри компании, а другие передавались на сторону.
- Разделить команды разработки и верификации ПО, обычно разработку оставить внутри компании, а верификацию передать на сторону. Передать на сторону и разработку, и верификацию ПО, но использовать внутренних экспертов для надзора и контроля за работой.

При решении, что именно будет передано на сторону, а что останется внутри компании, учитывайте интеллектуальную собственность, ключевую функциональность, которая должна работать быстро, интерфейсы между различными компонентами и области, хорошо замет-

ные заказчику. Такие вещи обычно лучше сохранять внутри компании.

Рекомендация 7: Определите наилучший подход к аутсорсингу. Как отмечалось в рекомендации 6, возможны многие подходы к аутсорсингу. Можно передать на сторону целую систему, а можно поручить поставщику только написание и выполнение тестов. Можно работать с местной компанией, а можно с компанией на другом конце света. Можно строить отношения с поставщиком на контрактной основе, а можно вообще владеть офшорной дочерней структурой. Мезак выделяет шесть распространенных стратегий разработки ПО [3]: (1) внутренние инженеры, (2) контрактный аутсорсинг внутри страны, (3) контрактный офшорный аутсорсинг, (4) смешанная модель из внутренних и офшорных ресурсов, (5) офшорная дочерняя структура, (6) build, operate, transfer (построить, эксплуатировать, передать), когда начинают с контрактной команды с возможностью в будущем перевести инженеров в дочернюю структуру. Поскольку лучший подход бывает разным, нужно оценить варианты, чтобы определить наилучшее соответствие как краткосрочным, так и долгосрочным потребностям организации.

Рекомендация 8: Выбирайте поставщика с умом. Это должно быть очевидно; однако, увидев несколько неудачных выборов, считаю полезным еще раз об этом напомнить. После того как определены сильные и слабые стороны, а также потребности команды заказчика, ищите поставщика, который соответствует этим потребностям и дополняет команду заказчика. Применяйте ту же тщательность и те же ожидания, что и при найме нового штатного сотрудника на ключевую должность: проверяйте рекомендации и рассматривайте резюме. Ниже перечислены некоторые конкретные области, которые следует учитывать:

- Опыт работы над проектами критичного по безопасности ПО. Опыт работы с DO-178C и/или DO-178B. Сходство прошлого опыта с планируемыми задачами. Языковые возможности, письменные и устные. Понимание предметной области безопасности. Знакомство с планируемым набором инструментов и средой. Готовность следовать установленным процессам. Стабильность команды и глубина кадрового состава.
- Способность выполнить порученные задачи в отведенное время. Способность добавить ресурсы, если проект пойдет не по плану.
- Опыт в конкретной предметной области, например навигация, связь, управление полетом.
- Репутация в отрасли, проверяйте рекомендации, и не только те, которые дал сам поставщик.
- Опыт работы с проектами планируемого масштаба, например малыми, средними или крупными.

- Перекрытие рабочего дня с командой заказчика.
- Образование, получите резюме и убедитесь, что есть средства обеспечить сохранение этой команды на проекте.
- Местоположение, насколько оно безопасно, защищено, удобно для доступа и так далее.
- Возможность при необходимости поддерживать работу на площадке заказчика. Это может включать анализ процесса получения виз, вариантов поездок, длительного проживания и так далее.
- Уважение к интеллектуальной собственности и закрытым данным.
- Вероятность уязвимостей безопасности; некоторые страны воспринимаются как источники вредоносных действий с государственной поддержкой.
- Оборудование, качественная телекоммуникационная система, стабильное электропитание, высокоскоростной Интернет, качество офисных помещений и оборудования, возможности видеоконференций, лабораторные помещения и так далее.
- Культурная совместимость команд поставщика и заказчика.
- Налаженная гарантия качества.
- Риски, например стабильность правительства, безопасность, конфликты интересов, отключения электроэнергии.

В книге *The Outsourcing Handbook* приводятся следующие шесть распространенных ошибок при выборе поставщика [1]:

1. Пожертвовать анализом потребностей ради эффектного поставщика.
2. Оценивать поставщика так, будто решающим фактором является экономия стоимости.
3. Плохо оценивать риски, связанные с поставщиком.
4. Слишком спешить в процессе выбора поставщика.
5. Недостаточно внимательно управлять взаимодействием между поставщиками.
6. Не поддерживать баланс между использованием текущих и новых поставщиков.

Помните об этом при оценке и выборе партнера по аутсорсингу. Этого нельзя подчеркнуть слишком сильно: не выбирайте только по цене. Почти всегда это заканчивается сожалениями.

Рекомендация 9: Настаивайте на сохранении одной и той же команды на всем протяжении проекта. Некоторые поставщики начинают с сильной и зрелой команды, но вскоре после запуска проекта этих инженеров переводят на другой проект. Обязательно зафикси-

руйте соглашение о составе команды письменно и проверяйте его соблюдение на протяжении всего проекта. Некоторую текучесть кадров следует считать нормальной, и иногда действительно бывает нужно перераспределить инженеров, чтобы подобрать более удачное соответствие. Однако состав команды важно контролировать постоянно.

Рекомендация 10: Подготовьте команду заказчика. Команде заказчика потребуется особое внимание. Некоторые инженеры могут быть настроены против аутсорсинга, а особенно против офшоринга. Убедитесь, что они понимают причины аутсорсинга, и постоянно подчеркивайте, что именно они критически важны для успеха проекта и компании. В зависимости от местоположения поставщика может потребоваться сдвигать графики, чтобы обеспечить перекрытие с его командой. Кроме того, и для команды поставщика, и для команды заказчика может понадобиться некоторое обучение культурной чувствительности и разнообразию. На протяжении всего проекта прислушивайтесь к обратной связи и относитесь к ней серьезно. В большинстве случаев у инженеров есть обоснованные замечания, которые стоит учитывать. Со временем станет ясно, чьим замечаниям можно надежно доверять, а чьи требуют определенной фильтрации; однако ценить нужно всех и давать каждому возможность высказываться свободно.

Рекомендация 11: Обеспечьте взаимное погружение команд. В зависимости от размера и характера проекта часто полезно, чтобы часть персонала заказчика работала на площадке поставщика, а часть персонала поставщика – на площадке заказчика. Лучше всего, по-видимому, работают краткосрочные пребывания, три месяца или меньше. Такое взаимное погружение полезно для обеих команд. Оно помогает лучше понимать проблемы проекта и упреждающе их решать.

Рекомендация 12: Начинайте с малого. Обычно лучше всего начинать аутсорсинговое приключение с небольших, хорошо определенных задач. Такие задачи помогают выстроить отношения, отладить процессные вопросы, выявить лидеров и нарастить экспертизу и уверенность. Многие компании начинают с пилотного проекта, как правило, это небольшая некритичная функция или инструмент. Это позволяет проверить коммуникационные способности, протестировать навыки технической команды и реализовать небольшую, но нужную функцию.

Независимо от того, выполняется пилотный проект или нет, все равно разумно начинать с малого. Некоторые компании сначала передают на сторону ПО более низких уровней, например уровня C или D. Если это проходит успешно, они могут передать на сторону более критичные и более крупные проекты.

Рекомендация 13: Четко задавайте ожидания. Работы, которые должны выполняться, и данные, которые должны создаваться, нужно документировать в описании объема работ

(statement of work, SOW). Описание объема работ должно быть полным, недвусмысленным, непротиворечивым и измеримым. Артефакты DO-178C должны быть включены в описание объема работ. Полезно, чтобы описание объема работ рассмотрели техническая команда и человек, хорошо разбирающийся в DO-178C и сертификации.

Рекомендация 14: Для расширения штата используйте общие инструменты и методы. Если аутсорсинговая команда будет расширять штат заказчика, рекомендуется использовать общие инструменты и методы. Общими должны быть следующие инструменты: инструмент или инструменты управления и фиксации требований, инструмент или инструменты проектирования, инструмент или инструменты работы с исходным кодом, инструмент или инструменты управления исходным кодом и инструмент или инструменты регистрации сообщений о проблемах. Также важно включить подробные процедуры, примеры и обучение. Иногда процедуры понятны команде заказчика, которая пользуется ими уже несколько лет, но могут быть непонятны внешней команде поставщика. Процедуры, которые должен использовать поставщик, возможно, придется усилить.

При передаче на сторону целого компонента или системы использовать тот же набор инструментов и процессов может быть необязательно. Однако важно (1) понимать процессы поставщика, (2) обеспечить, чтобы поставщик либо передавал, либо поддерживал данные и среду в долгосрочной перспективе в соответствии с требованиями заказчика и сертифицирующих органов, и (3) спланировать и реализовать стратегию перехода для передачи продукта и данных.

Рекомендация 15: Разработайте план управления поставщиком. Приказ Федерального управления гражданской авиации США 8110.49, изменение 1, требует документированного плана управления поставщиком [2].

Такой план может быть частью Плана сертификации ПО (PSAC) или системного плана сертификации, а может существовать как общеорганизационный план. План управления поставщиком должен обеспечивать соблюдение поставщиками, включая расширение штата и офшорные дочерние команды, всех нормативных требований, политик, руководящих указаний, соглашений и стандартов, применимых к проекту. В плане также объясняется, как именно будут управляться все поставщики. Для сертифицирующих органов особый интерес представляют соблюдение требований, интеграция различных артефактов, например требований, проекта ПО и кода, взаимодействие с сертифицирующим органом, регистрация сообщений о проблемах и их устранение, интеграция, верификация, управление конфигурацией, подтверждение соответствия и хранение данных [2]. Сертифицирующие органы хотят понимать, что именно делается, кем и где. Они также хотят видеть, как именно аутсорсинг управляется со стороны заявителя, то есть стороны, подающей заявку на сертификацию или

разрешение. Даже если офшорная площадка принадлежит заказчику или заявителю либо является внутренней командой, используемой для *расширения штата*, а не как *поставщик*, это управление все равно хочет понимать, как этим усилием управляют.*

Рекомендация 16: Согласуйте процесс регистрации сообщений о проблемах. Разработчики и верификаторы ПО могут не осознавать влияние низкоуровневой проблемы на безопасность системы или воздушного судна. Поэтому для всех поставщиков рекомендуется единая категоризация сообщений о проблемах. Кроме того, на протяжении всей разработки и/или верификации необходима прозрачность по сообщениям поставщика о проблемах; ждать до конца проекта слишком рискованно. Любые сообщения о проблемах, перенесенные за пределы первоначальной сертификации, потребуют обоснования и рассмотрения всеми заинтересованными сторонами, включая сертифицирующие органы.

Рекомендация 17: Обучайте по ключевым направлениям. В начале программы разумно определить потребности в обучении для всей команды, и заказчика, и поставщика. Часто дефицит наблюдается в таких областях, как соответствие DO-178B/C, тестирование на робастность, фиксация требований низкого уровня, документирование воспроизводимых анализов и процедур, интеграция, а также анализ межкомпонентной связности по данным и по управлению. Для больших команд может потребоваться изобретательность, чтобы нужное обучение дошло до нужных участников. Распространенная стратегия состоит в том, чтобы сначала обучить технических лидеров, а затем дать им возможность передать знания дальше. Полезным может быть и компьютерное обучение с разнообразными примерами. Кроме того, регулярно обновляемое онлайн-руководство по лучшим практикам может стать отличным способом донести знания до широкой аудитории.†

Рекомендация 18: Постоянно уделяйте внимание коммуникации. Плохая коммуникация – один из главных факторов провала проекта, поэтому самой схеме коммуникации и ее эффективности нужно уделять особое внимание. Как пишет Мезак: «Использование аутсорсинга похоже на брак. Чтобы отношения работали, нужна вовлеченность обеих сторон. Нужна хорошая коммуникация» [3]. Ниже приведено несколько способов усилить коммуникацию:

- Проводите очные встречи для лидеров команд, особенно в начале и в ключевые моменты проекта.
- Организуйте частые телеконференции и/или виртуальные встречи между командами. Если поставщик офшорный, это может быть ежедневная телеконференция во время перекрытия смен. Видеоконференции и сетевой совместный доступ к документам могут быть полезны.
- Поощряйте команды узнавать друг друга не только профессионально, но и лично.

- Используйте графику для объяснения сложных концепций. Диаграмма часто помогает снять коммуникационные проблемы.
- Обеспечьте некоторый уровень очного присутствия на площадке, то есть взаимное погружение команд. Поощряйте немедленное сообщение о проблемах и опасениях, а не ожидание, пока они выйдут из-под контроля.
- Разработайте способ эскалации на случай, если одна из компаний считает, что существенная проблема, техническая или программная, не получает должного внимания.
- Постоянно ищите способы улучшения коммуникации на основе извлеченных уроков.

Рекомендация 19: Оценивайте и снижайте риски. Разработка программного обеспечения содержит множество рисков. Аутсорсинг может увеличить часть из них. Как уже отмечалось, квалификация поставщика должна быть тщательно оценена до его выбора. Любые обнаруженные недостатки следует фиксировать как риск. По мере развития проекта риски нужно последовательно переоценивать и определять меры по их снижению. Обнаружить риск и ничего не делать с ним почти бесполезно. Необработанные риски похожи на дыры в днище лодки. Если ими не заниматься, они могут потопить лодку, а заодно и проект. Непрерывная оценка рисков с конкретными мерами снижения помогает избежать такого затопления.

Рекомендация 20: Учитывайте чрезмерную нагрузку. С оценкой рисков тесно связано снижение риска *чрезмерной нагрузки*. Как отмечалось ранее, офшорный поставщик в стране без двусторонних соглашений может стать чрезмерной нагрузкой для правительства США. Поэтому Федеральное управление гражданской авиации США должно понимать, что за поставщиком ведется надлежащий надзор и что риски заранее прорабатываются. Ниже приведены некоторые распространенные стратегии, позволяющие избежать чрезмерной нагрузки:

- Ограничить офшорные работы верификацией, а тестирование для получения сертификационного зачета выполнять в США.
- Привлекать американскую команду, то есть команду заказчика, к экспертным рассмотрением данных. Использовать уполномоченных представителей Федерального управления гражданской авиации США для проведения выездных аудитов работ поставщика, чтобы убедиться в их соответствии требованиям.
- Ограничивать офшорные работы более низкими уровнями ПО до тех пор, пока сертифицирующие органы не будут уверены и в заказчике, и в поставщике.

Рекомендация 21: Хорошо этим управляйте. Многие аутсорсинговые катастрофы, которые мне доводилось наблюдать, были следствием плохого управления. Аутсорсинговыми про-

ектами должны руководить квалифицированные и опытные специалисты. Руководители в такой роли должны обладать следующими навыками и способностями: умением принимать хорошие решения в условиях неопределенности, готовностью принимать изменения, хорошими навыками выстраивания отношений и коммуникации, сильными навыками управления знаниями и настойчивостью [1]. Кроме того, технический руководитель должен обладать сильными техническими навыками и быть способен понимать технические проблемы и риски.

Полный перечень контрольных точек с детальным отслеживанием и приемкой этих точек является важным инструментом управления любым программным проектом, включая аутсорсинговый. Необходимо выявлять зависимости между командами и задачами и обеспечивать достоверность статуса. График и контрольные точки следует обновлять по мере необходимости на основе реального хода проекта.

Рекомендация 22: Будьте реалистичны. Технические ожидания и ожидания по графику должны быть реалистичными. Нереалистично ожидать, что команда из новых инженеров быстро начнет выдавать качественную работу. В общем случае разумно надеяться на лучшее, но планировать по худшему варианту. Планирование, основанное только на успехе, никому не помогает. В какой-то момент оно может сделать всех довольными, но если оно нереалистично, это довольство продлится недолго. Хорошие руководители работают с реальностью. Иоганна Ротман пишет: «Планируйте так, будто каждый проект займет больше времени и потребует больше затрат, особенно в начале отношений по аутсорсингу. Мое практическое правило – увеличить расчетное время на 30 процентов для первого проекта. Затем отслеживайте проект, чтобы понять, нужно ли увеличить эту оценку» [4].

Рекомендация 23: Держите в уме долгосрочные цели. Некоторые компании разочаровываются, потому что их аутсорсинговые усилия дают не тот результат, который им представлялся. Важно помнить о долгосрочных целях. Если ни краткосрочные, ни долгосрочные цели не выглядят реалистично, возможно, их нужно пересмотреть.

Рекомендация 24: Планируйте надзор. Сертифицирующие органы требуют прозрачности в отношении подхода к надзору, включая управленческий надзор, технический надзор, надзор по гарантии качества, меры защиты и надзор за взаимодействием с сертифицирующим органом. Каждая из этих областей должна быть подробно описана в Плане сертификации ПО (PSAC) и/или в каком-либо другом плане, доступном сертифицирующим органам.

Рекомендация 25: Планируйте послепроектную поддержку. В аутсорсинговой схеме нужно заранее определить, кто будет выполнять поддержку после завершения проекта. Иными словами, кто будет сопровождать данные после первоначальной сертификации? Это повлияет на распределение ролей и обязанностей и на то, какие данные и куда должны быть пе-

реданы. Например, если поставщик, разработавший тестовые примеры и процедуры, после сертификации использоваться не будет, то важно обеспечить передачу всех инструментов, данных, оборудования и прочего, чтобы в будущем можно было обновлять тестовые данные.

Тот, кто будет заниматься поддержкой, должен до внесения изменений получить все данные ЖЦ ПО, включая открытые сообщения о проблемах, указатель конфигурации ПО и итоговый отчет разработки ПО.

Рекомендация 26: Внедряйте непрерывное совершенствование. Каждый завершенный аутсорсинговый проект дает множество извлеченных уроков. Важно фиксировать эти уроки, учиться на них и соответствующим образом корректировать процессы.

26.5 Резюме

Аутсорсинг никуда не денется, поэтому важно научиться внедрять его и управлять им эффективно. В этой главе были определены распространенные проблемы и приведены некоторые рекомендации, помогающие предотвращать или решать такие проблемы. Пожалуйста, обращайтесь и к другим источникам, и к экспертам в данной области. Это всего лишь взгляд одного человека. Удачи и приятного опыта!

Литература

1. M. J. Powers, K. C. Desouza и C. Bonifazi, *The Outsourcing Handbook: How to Implement a Successful Outsourcing Process* (Philadelphia, PA: Kogan Page, 2006).
2. Federal Aviation Administration, *Software Approval Guidelines*, Order 8110.49 (Washington, DC: Federal Aviation Administration, Change 1, September 2011).
3. S. Mezak, *Software Without Borders* (Los Altos, CA: Earthrise Press, 2006).
4. J. Rothman, 11 steps to successful outsourcing: A Contrarian's view, *Computer World* (article #84847, 2003).
 - http://www.computerworld.com/s/article/84847/11_Steps_to_Successful_Outsourcing_A_Contrarian's_View (дата обращения: август 2011 г.).

*Двустороннее соглашение должно охватывать тот вид работ, который делегируется. Частичные или ограниченные двусторонние соглашения могут не распространяться на программное обеспечение или современную авионику.

*Английский язык является международным языком, используемым авиационным сообществом. Поскольку большинство воздушных судов сертифицируется для международной эксплуатации, английский обычно обязателен, особенно для проектов, проходящих сертификацию FAA.

*Напомним из главы 9, что по-настоящему успешная программа тестирования – это та, которая находит ошибки, а не та, которая проходит все тесты без замечаний.

*Этот тест можно найти на сайте www.softwarewithoutbordersbook.com.

*Скобки добавлены для пояснения.

*В Certification Memo CM-SWCEH-002 агентства EASA содержатся сходные указания.

†Такое руководство следует описать в планах ПО и держать под управлением конфигурацией.

Приложение А. Пример критериев перехода

Сокращения

Сокращение	Расшифровка
CM	Управление конфигурацией
FDAL	Уровень гарантии разработки функции
PR	Сообщение о проблеме
PSAC	План сертификации ПО
SCI	Указатель конфигурации ПО
SDD	Описание проекта ПО
SDP	План разработки ПО
SECI	Указатель конфигурации среды ЖЦ ПО
SQA	Гарантия качества ПО
SRD	Документ системных требований
SVP	План верификации ПО
SWRD	Документ требований высокого уровня к ПО (Software Requirements Document)

Обзор

В таблицах А.1 и А.2 приведен пример критериев перехода, которые могут быть включены в План разработки ПО (таблица А.1) и План верификации ПО (таблица А.2).[*] Критерии Плана разработки ПО (таблица А.1) дают только высокоуровневое представление о рассмотрении, поскольку в Плане верификации ПО (таблица А.2) для этих рассмотрений приведены подробные критерии перехода.

Таблица А.1 Пример критериев перехода для Плана разработки ПО

Фаза	Входные критерии	Мероприятия	Выходные критерии
Разработка и рассмотрение требований к ПО	Определены системный уровень гарантии разработки функции и уровень ПО Выпущен План сертификации системы Выпущен План сертификации ПО Выпущен План разработки ПО Выпущены стандарты требований к ПО Системные требования достаточно зрелые, чтобы начать декомпозицию	Документировать требования высокого уровня к ПО (в Документе требований высокого уровня к ПО) на основе Документа системных требований для требований, распределенных на ПО Выявить требования, влияющие на безопасность (пометить атрибутом безопасности) Выявить все производные требования к ПО (требования, которые не трассируются на требования из Документа системных требований и задают детали реализации) и привести обоснование, зачем нужно каждое из них Разработать двунаправленную трассируемость между Документом системных требований и Документом требований высокого уровня к ПО Включить в Документ требований высокого уровня к ПО интерфейсы/взаимодействия по мере необходимости Включить в Документ требований высокого уровня к ПО ссылку на аппаратурные данные, если это применимо Документировать любые отклонения от стандартов требований и получить одобрение гарантии качества ПО Документировать требования робастности (пометить как требования робастности) Рассмотреть Документ требований высокого уровня к ПО по процедурам компании, определенным в Плане верификации ПО, и при необходимости внести обновления Провалидировать производные требования (как часть рассмотрения), чтобы убедиться, что это правильные требования (корректные и полные) Привлекать персонал по безопасности к рассмотрению всех производных требований Документировать среду разработки в Указателе конфигурации среды ЖЦ ПО	Документ системных требований выпущен и находится под управлением конфигурацией Документ требований высокого уровня к ПО рассмотрен, обновлен, выпущен и находится под управлением конфигурацией Производные требования высокого уровня к ПО рассмотрены соответствующим персоналом по системной безопасности Указатель конфигурации среды ЖЦ ПО подготовлен в черновике и находится под управлением конфигурацией
Разработка и рассмотрение Описания проекта ПО	Выпущен План сертификации ПО Выпущен План разработки ПО Выпущены стандарты проектирования ПО Документ системных требований и Документ требований высокого уровня к ПО достаточно зрелые, чтобы начать проектирование	Документировать требования низкого уровня, используя Документ системных требований, Документ требований высокого уровня к ПО и другие указанные требования в качестве входных данных Выявить интерфейсы и взаимодействия в Описании проекта ПО по мере необходимости Выявить все производные требования низкого уровня (требования, которые не трассируются на требования из Документа требований высокого уровня к ПО и представляют собой детали реализации) и привести обоснование, зачем нужно каждое из них Документировать требования робастности (пометить как требования робастности) Провалидировать производные требования (как часть рассмотрения), чтобы убедиться, что это правильные требования, то есть что они корректны и полны (подробности рассмотрения см. в Плане верификации ПО) Разработать двунаправленную трассируемость между требованиями высокого уровня и требованиями низкого уровня Разработать архитектуру ПО так, чтобы она соответствовала требованиям Рассмотреть Описание проекта ПО по процессу, определенному в Плане верификации ПО, и внести необходимые обновления Документировать любые отклонения от стандартов проектирования и получить одобрение гарантии качества ПО Привлекать персонал по безопасности к рассмотрению всех производных требований	Описание проекта ПО, включая требования низкого уровня и архитектуру, рассмотрено, обновлено, выпущено и находится под управлением конфигурацией Документ системных требований и Документ требований высокого уровня к ПО выпущены и находятся под управлением конфигурацией Производные требования низкого уровня рассмотрены соответствующим персоналом по системной безопасности
Разработка и рассмотрение кода	Выпущен План сертификации ПО Выпущен План разработки ПО Выпущены стандарты кодирования Документ требований высокого уровня к ПО и Описание проекта ПО доступны и достаточно зрелые, чтобы начать кодирование	Писать код, используя Документ требований высокого уровня к ПО, Описание проекта ПО, План разработки ПО и стандарты кодирования как входные данные Разработать двунаправленную трассируемость между кодом и требованиями низкого уровня Рассмотреть код и обновить его по замечаниям (подробности рассмотрения см. в Плане верификации ПО) Документировать любые отклонения от стандартов кодирования и получить одобрение гарантии качества ПО Разработать и рассмотреть Указатель конфигурации ПО	Код рассмотрен, обновлен, выпущен и находится под управлением конфигурацией Документ системных требований, Документ требований высокого уровня к ПО и Описание проекта ПО выпущены и находятся под управлением конфигурацией Указатель конфигурации ПО рассмотрен и находится под управлением конфигурацией

Фаза	Входные критерии	Мероприятия	Выходные критерии
Интеграция кода	Код доступен Указатель конфигурации ПО подготовлен в черновике Параметры компилятора и компоновщика указаны в Указателе конфигурации среды ЖЦ ПО	Разработать процедуры сборки (в Указателе конфигурации ПО) и дать независимому лицу выполнить их для рассмотрения повторяемости Собрать код, используя процедуры сборки До формального тестирования скомпилировать и скомпоновать выпущенный код, используя утвержденные процедуры сборки (в Указателе конфигурации ПО) Разработать процедуры загрузки (в Указателе конфигурации ПО) и дать независимому лицу выполнить их для рассмотрения повторяемости Обновить и рассмотреть Указатель конфигурации ПО, включая процедуры сборки и загрузки Обновить (при необходимости) и рассмотреть Указатель конфигурации среды ЖЦ ПО (до формальной сборки кода)	Код выпущен и находится под управлением конфигурацией Указатель конфигурации ПО, включая процедуры сборки и процедуры загрузки, и Указатель конфигурации среды ЖЦ ПО рассмотрены, выпущены и находятся под управлением конфигурацией Выпущенный код скомпилирован и скомпонован Исполняемый объектный код выпущен и находится под управлением конфигурацией

Таблица А.2 Пример критериев перехода для Плана верификации ПО

Этап	Входные критерии	Мероприятия	Выходные критерии
Плановые рассмотрения	• Все пять планов находятся под управлением конфигурацией. • Стандарты требований, проектирования и кодирования выпущены и находятся под управлением конфигурацией.	• Проверить все планы и стандарты по контрольным перечням для планирования. • Задokumentировать замечания по рассмотрению. • Обновить планы и стандарты по замечаниям. • Закрывать или разрешить все пункты доработки и сообщения о проблемах, выявленные при рассмотрении. • Выпустить обновленные планы и стандарты.	• Все планы и стандарты рассмотрены, обновлены, выпущены и находятся под управлением конфигурацией. • Все пункты доработки и сообщения о проблемах по результатам рассмотрения закрыты или разрешены.

Этап	Входные критерии	Мероприятия	Выходные критерии
Рассмотрение Документа требований высокого уровня к ПО	• Планы и стандарты требований выпущены. • Системные требования выпущены. • Документ требований высокого уровня к ПО находится под управлением конфигурацией. Если документ рассматривается по разделам, каждый раздел должен быть достаточно зрелым и находится под управлением конфигурацией до рассмотрения.	• Рассмотреть Документ требований высокого уровня к ПО относительно системных требований, используя контрольный перечень для Документа требований высокого уровня к ПО из Плана верификации ПО. • Валидировать производные требования. • Убедиться, что соблюдены стандарты требований, определенные в Плане разработки ПО. • Подтвердить корректность и полноту двунаправленной трассируемости между Документом требований высокого уровня к ПО и Документом системных требований. • Убедиться, что Документ требований высокого уровня к ПО надлежащим образом обновлен по замечаниям рассмотрения. • Создать пункт доработки до установления базовой версии или сообщение о проблеме после установления базовой версии, если найдены проблемы в связанных данных под управлением конфигурацией. • Закрывать или разрешить выявленные пункты доработки и сообщения о проблемах. • Убедиться, что весь набор производных требований рассмотрен соответствующим персоналом по системной безопасности.	• Документ требований высокого уровня к ПО обновлен по результатам рассмотрения, выпущен и находится под управлением конфигурацией. • Все пункты доработки и сообщения о проблемах по рассмотрению закрыты или разрешены.

Этап	Входные критерии	Мероприятия	Выходные критерии
Рассмотрение проекта ПО	• Документ системных требований и Документ требований высокого уровня к ПО рассмотрены, обновлены и выпущены. • Описание проекта ПО находится под управлением конфигурацией. • Стандарты проектирования выпущены. • Трассируемость между требованиями высокого уровня и требованиями низкого уровня находится под управлением конфигурацией.	• Рассмотреть Описание проекта ПО по контрольному перечню проектирования из Плана верификации ПО. • Валидировать производные требования. • Убедиться, что соблюдены стандарты проектирования, указанные в Плана разработки ПО, как для требований низкого уровня, так и для архитектуры. • Подтвердить корректность двунаправленной трассируемости между требованиями высокого уровня и требованиями низкого уровня. • Убедиться, что Описание проекта ПО корректно обновлено по замечаниям рассмотрения. • Создать пункт доработки до установления базовой версии или сообщение о проблеме после установления базовой версии, если найдены проблемы в связанных данных под управлением конфигурацией. • Закрыть или разрешить выявленные пункты доработки и сообщения о проблемах. • Убедиться, что производные требования рассмотрены соответствующим персоналом по системной безопасности.	• Описание проекта ПО обновлено по результатам рассмотрения, выпущено и находится под управлением конфигурацией. • Все пункты доработки и сообщения о проблемах по рассмотрению закрыты или разрешены.

Этап	Входные критерии	Мероприятия	Выходные критерии
Разработка и рассмотрение Описания проекта ПО	• План сертификации ПО выпущен. • План разработки ПО выпущен. • Стандарты проектирования ПО выпущены. • Документ системных требований и Документ требований высокого уровня к ПО достаточно зрелые, чтобы начать проектирование.	• Документировать требования низкого уровня, используя Документ системных требований, Документ требований высокого уровня к ПО и другие указанные требования как входные данные. • Выявить интерфейсы и взаимодействия в Описании проекта ПО по мере необходимости. • Выявить производные требования низкого уровня и задокументировать обоснование для каждого из них. • Документировать требования робастности. • Валидировать производные требования, чтобы убедиться в их корректности и полноте. • Разработать двунаправленную трассируемость между требованиями высокого уровня и требованиями низкого уровня. • Разработать архитектуру ПО, согласованную с требованиями. • Рассмотреть Описание проекта ПО по процессу, определенному в Плане верификации ПО, и внести нужные обновления. • Документировать отклонения от стандартов проектирования и получить одобрение гарантии качества ПО. • Привлечь персонал по безопасности к рассмотрению всех производных требований.	• Описание проекта ПО, включая требования низкого уровня и архитектуру, рассмотрено, обновлено, выпущено и находится под управлением конфигурацией. • Документ системных требований и Документ требований высокого уровня к ПО выпущены и находятся под управлением конфигурацией. • Производные требования низкого уровня рассмотрены соответствующим персоналом по системной безопасности.
Разработка тестовых примеров и процедур	• План верификации ПО выпущен. • Инструкции по сборке тестов находятся под управлением конфигурацией. • Тесты неформально прогнаны разработчиком тестов, и результаты доступны для рассмотрения.	• Выявить тестовые примеры на робастность, если требование помечено как требование робастности; если такие тесты не нужны, задокументировать обоснование. • Убедиться, что инструменты, требующие квалификации для выполнения тестов, квалифицированы или достаточно стабильны, чтобы начать разработку тестов. • Разработать двунаправленные данные трассировки между тестовыми примерами и требованиями, а также между тестовыми примерами и тестовыми процедурами. • Задокументировать инструкции по сборке тестов и убедиться, что все необходимые для сборки и выполнения тестов инструменты описаны в Указателе конфигурации среды ЖЦ ПО. • Добавить сведения о тестах в Указатель конфигурации ПО. • Рассмотреть тестовые примеры и процедуры и обновить их по мере необходимости.	• Указатель конфигурации среды ЖЦ ПО выпущен. • Указатель конфигурации ПО обновлен информацией о тестовых процедурах и находится под управлением конфигурацией.

Этап	Входные критерии	Мероприятия	Выходные критерии
Разработка интеграционных анализов	• План верификации ПО выпущен. • Данные, необходимые для анализа, достаточно зрелые, чтобы начать анализ.	• Документировать процедуры анализа, используя критерии, определенные в Плане верификации ПО или другом применимом документе. • Проверить процедуры на повторяемость и соответствие установленным критериям. • Обновить процедуры по замечаниям. • Если в анализе используются квалифицируемые инструменты, убедиться, что их квалификация завершена. • Выполнить дополнительные анализы, например анализ компоновки и загрузки, анализ наихудшего времени выполнения и анализ карты памяти, используя выпущенные процедуры. • Задокументировать результаты анализа в отчете. • Задокументировать любые проблемы, выявленные во время анализа, в сообщениях о проблемах.	• Процедуры анализа рассмотрены, обновлены и выпущены. • Результаты анализа задокументированы в отчете и находятся под управлением конфигурацией. • По всем проблемам, выявленным при анализе, заведены сообщения о проблемах.
Рассмотрение тестовых примеров и процедур	• Документ системных требований, Документ требований высокого уровня к ПО и Описание проекта ПО выпущены. • Применимые тестовые примеры и процедуры находятся под управлением конфигурацией. • Указатель конфигурации среды ЖЦ ПО и Указатель конфигурации ПО находятся под управлением конфигурацией. • Данные трассировки между тестовыми примерами и требованиями, а также между тестовыми примерами и тестовыми процедурами, разработаны и находятся под управлением конфигурацией.	• Рассмотреть тестовые примеры и процедуры по контрольному перечню из Плана верификации ПО. • Убедиться в корректности двунаправленной трассируемости между тестовыми примерами и требованиями, а также между тестовыми примерами и тестовыми процедурами. • Убедиться, что все требования охвачены тестированием. • Убедиться, что для тестов определены ожидаемые результаты и критерии прохождения/непрохождения. • Выполнить инструкции по сборке тестов для проверки их корректности. • Рассмотреть неформальные результаты тестов и убедиться, что тесты выполнимы, проходят успешно, а результаты корректно задокументированы. • Убедиться, что требования робастности учтены, либо имеется обоснование их отсутствия. • Закрыть или разрешить пункты доработки и сообщения о проблемах, найденные при рассмотрении.	• Тестовые примеры и процедуры обновлены по результатам рассмотрения и выпущены. • Все пункты доработки и сообщения о проблемах по рассмотрению закрыты или разрешены.

Этап	Входные критерии	Мероприятия	Выходные критерии
Выполнение тестов	• Тестовые примеры и процедуры выпущены. • Код инструмента и квалификационные данные выпущены. • Инструкции по сборке тестов выпущены. • Указатель конфигурации среды ЖЦ ПО выпущен. • Все тесты прошли пробный прогон. • Выполнено рассмотрение готовности к тестам и удовлетворены критерии входа.	• Уведомить специалистов по гарантии качества ПО и специалистов по взаимодействию с сертифицирующим органом о намерении выполнить тесты. • Выполнить тесты. • Проанализировать результаты тестов и определить результат прохождения или непрохождения. • Выполнить анализ структурного покрытия и задокументировать результаты. • Выявить любые непрохождения тестов, при необходимости повторно выполнить тесты и задокументировать проблемы в сообщениях о проблемах. • Проанализировать выявленные непрохождения тестов, чтобы понять, требуется ли доработка. • Задокументировать результаты в тестовых отчетах. • Проанализировать недостающее структурное покрытие и по мере необходимости завести сообщения о проблемах.	• Результаты тестов находятся под управлением конфигурацией. • Для непрошедших тестов и всех правок заведены сообщения о проблемах.
Рассмотрение результатов тестирования	• Тестовые примеры и процедуры выпущены. • Тестовые примеры и процедуры выполнены. • Результаты тестов задокументированы в тестовом отчете. • Указатель конфигурации ПО обновлен сведениями о тестах. • Тестовый отчет находится под управлением конфигурацией.	• Рассмотреть тестовый отчет на полноту и корректность по контрольному перечню из Плана верификации ПО. • Подтвердить корректность и полноту трассируемости между тестовыми процедурами и результатами тестирования. • Убедиться, что все непрохождения тестов объяснены и по ним открыты сообщения о проблемах. • Убедиться, что правки задокументированы и одобрены гарантией качества ПО. • Закрыть или разрешить пункты доработки и сообщения о проблемах по результатам рассмотрения. • Проверить Указатель конфигурации ПО на корректность тестовой информации и обновить его при необходимости.	• Тестовый отчет рассмотрен, обновлен и выпущен. • Все пункты доработки и сообщения о проблемах по рассмотрению закрыты или разрешены. • Указатель конфигурации ПО рассмотрен, обновлен и выпущен.

Этап	Входные критерии	Мероприятия	Выходные критерии
Рассмотрение результатов анализа	- Анализируемые данные выпущены. - Процедура анализа выпущена. - Результаты анализа содержатся в отчете и находятся под управлением конфигурацией. - Для используемых в анализе инструментов, если требуется, завершена квалификация. - Указатель конфигурации ПО обновлен сведениями об анализе.	- Рассмотреть отчет по анализу и убедиться, что анализы выполнены в соответствии с процедурами. - Убедиться, что анализ выполнен с требуемой независимостью. - Подтвердить, что результаты анализа согласуются с ожидаемыми результатами. - Убедиться, что версии используемых инструментов совпадают с квалифицированными версиями, если это применимо. - Проверить все связанные сообщения о проблемах и их обоснование на приемлемость. - Проверить Указатель конфигурации ПО на корректность сведений об анализе и обновить его при необходимости.	- Отчет по анализу рассмотрен, обновлен и выпущен. - Все пункты доработки и сообщения о проблемах по результатам рассмотрения закрыты или разрешены. - Указатель конфигурации ПО рассмотрен, обновлен и выпущен.

Эти критерии перехода приведены как пример и не предназначены для использования в сертификации, поскольку в них представлена только часть ЖЦ ПО, предлагаемая в Плане разработки ПО и Плане верификации ПО.

Приложение В. Проблемные области ОСРВ

Сокращения

Сокращение	Расшифровка
API	Интерфейс прикладного программирования
C	Мертвый или отключенный код
COTS	Готовое коммерческое ПО (commercial off-the-shelf)
CPU	Центральный процессор
D	Согласованность данных
FAA	Федеральное управление гражданской авиации США (Federal Aviation Administration)
I	Прерывания и исключения
I/O	Ввод/вывод
M	Доступ к памяти и устройствам ввода/вывода
Q	Очереди
RTOS	ОСРВ (операционная система реального времени, real-time operating system)
S	Планирование
T	Управление задачами
TCB	Блок управления задачами

Обзор

Это приложение содержит таблицу из исследовательского отчета Федерального управления гражданской авиации США (Federal Aviation Administration, FAA) DOT/FAA/AR-02/118, *Исследование готовых коммерческих операционных систем реального времени для применения в авиации (Study of Commercial Off-the-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications)*. Таблица В.1 общедоступна на сайте www.faa.gov; однако здесь она приведена для удобства. В ней указаны типовые проблемные области для операционной системы реального времени (real-time operating system, RTOS). Следует рассматривать следующие семь функциональных областей: согласованность данных (D), мертвый или отключенный код (C), управление задачами (T), планирование (S), доступ к памяти и устройствам ввода/вывода (M), очереди (Q), а также прерывания и исключения (I). Дополнительные сведения об операционной системе реального времени см. в главе 20.

Таблица В.1 Типовые проблемные области ОСРВ

Код	Функциональная область	Проблемная область
D1		Согласованность данных. Повреждение данных или их потеря самой ОСРВ. Данные, видимые ОСРВ, оказываются повреждены или «утрачены» самой ОСРВ.
D2		Согласованность данных. Повреждение входных данных или их потеря ОСРВ. ОСРВ некорректно обрабатывает входные данные или теряет их, неверно сохраняя либо присваивая неправильные значения переменным данных или возвращаемым результатам.
D3		Согласованность данных. Ошибочные данные или результаты из-за неверных вычислений или операций ОСРВ. Неправильные значения данных присваиваются переменным или возвращаются как результаты.
D4		Согласованность данных. Аномальные параметры. Вычисления библиотечных математических функций могут возвращать непредсказуемые малые числа, если переданные в качестве параметров значения аномальны.
C1		Мертвый или отключенный код. Включение отключенного кода. Неиспользуемые функции могут загружаться вместе с приложением, хотя они никогда не вызываются. Это также может зависеть от компоновщика или загрузчика, который компонует исполняемый объектный код в исполняемый образ и/или загружает образ в память целевого вычислителя. Непреднамеренная активация такого кода может иметь неизвестные эффекты и обычно приводит к отказу системы.
C2		Мертвый или отключенный код. Генерация мертвого кода. Компилятор или компоновщик может порождать дополнительное программное обеспечение, которое не проверяется при тестировании на основе требований и анализе покрытия. Это особенно опасно для приложений уровня А, где разработчик обязан «объяснить» исполняемый объектный код, не трассируемый на исходный код. Такой код может оказаться мертвым, а также не выполняться при тестировании на основе требований и не учитываться в анализе структурного покрытия, который обычно выполняется на уровне исходного кода. Объектный код, сгенерированный компилятором или компоновщиком, не освобождается от выполнения этих целей по соответствию требованиям и робастности для уровней А-D, а для требований низкого уровня уровня С.
T1		Управление задачами. Задача завершается или удаляется. Задача выполняется до завершения или удаляется другой задачей. Если программная модель требует, чтобы задача работала бесконечно в бесконечном цикле, вызов интерфейса прикладного программирования для удаления задачи следует убрать.
T2		Управление задачами. Переполнение области хранения ядра. Центральная область хранения в ядре, содержащая блоки управления задачами и другие объекты ядра, может переполниться из-за вредоносной задачи, постоянно выделяющей новые объекты ядра. Это может повлиять на выполнение других задач. Для защиты других задач следует вводить систему квот.
T3		Управление задачами. Превышен размер стека задачи. Стек задачи перезаписывается, что ведет к непредсказуемому поведению системы и повреждению данных стека.
S1		Планирование. Повреждение блоков управления задачами (task control block, TCB). Эти блоки могут быть повреждены, что нарушает операции планирования в операционной системе реального времени. Данные планирования должны быть защищены от доступа пользовательского ПО.
S2		Планирование. Чрезмерная блокировка задач из-за инверсии приоритетов. Пользовательская задача высокого приоритета может чрезмерно блокироваться задачей низкого приоритета, если они разделяют общий ресурс и промежуточная задача вытесняет низкоприоритетную.
S3		Планирование. Взаимная блокировка. Если двум задачам нужны одни и те же два ресурса, но они планируются в неправильной последовательности, это может вызвать взаимную блокировку, когда каждая задача блокирует другую.
S4		Планирование. Задачи порождают дополнительные задачи, истощающие ресурсы центрального процессора. Новые задачи, порожденные существующей задачей, могут нарушить планируемость всех задач в системе. Пользовательским приложениям не следует разрешать создавать новые задачи по собственному усмотрению.
S5		Планирование. Повреждение назначения приоритетов задач. Повышение или понижение приоритетов задач в системе может привести к тому, что задача перестанет планироваться или система перестанет своевременно реагировать. Возможность менять приоритет задачи следует ограничивать особыми случаями, например для предотвращения инверсии приоритетов.
S6		Планирование. Сервисные вызовы с неограниченным временем выполнения. На планируемость задач влияет наличие сервисных вызовов ядра с неограниченным временем выполнения. Это может сказаться и на времени выполнения самой задачи, и на накладных расходах ядра при переключении между задачами. Для сервисных вызовов ядра следует обеспечивать ограниченное время выполнения независимо от нагрузки системы.

Код	Функциональная область	Проблемная область
M1		Доступ к памяти и устройствам ввода-вывода. Фрагментация кучи. Выделение, освобождение памяти и «сборка мусора» в куче могут приводить к фрагментации свободной памяти, затруднять будущие выделения и делать временный анализ непредсказуемым. Динамическое распределение памяти, освобождение памяти и «сборка мусора» должны быть очень ограничены и строго контролироваться.
M2		Доступ к памяти и устройствам ввода-вывода. Некорректное разыменование указателя. Через сервисный вызов в ядро может быть передана некорректная ссылка на объект, например на семафор, что может привести к катастрофическим последствиям. Ядро должно проверять корректность ссылок указателей.
M3		Доступ к памяти и устройствам ввода-вывода. Перезапись данных. Данные записываются за пределами выделенных границ и перезаписывают или повреждают соседние данные других функций в памяти.
M4		Доступ к памяти и устройствам ввода-вывода. Нарушение когерентности кэша. Время доступа возрастает из-за промахов кэша. Это происходит, когда нужных данных нет в кэше и их приходится считывать из другой, обычно более медленной, памяти. Возможна потеря данных из-за пропущенных обновлений памяти.
M5		Доступ к памяти и устройствам ввода-вывода. Память может быть заблокирована или недоступна. Таблицы страниц блока управления памятью могут быть неверно настроены или повреждены так, что доступ к области памяти становится невозможен.
M6		Доступ к памяти и устройствам ввода-вывода. Несанкционированный доступ к критическим системным устройствам. Несанкционированный доступ к устройствам ввода-вывода может привести к неправильной работе системы. Ядро должно обеспечивать обязательный контроль доступа ко всем критическим устройствам.
M7		Доступ к памяти и устройствам ввода-вывода. Ресурсы не контролируются. Необходимо контролировать корректное выделение и использование ресурсов, иначе другой ресурс может оказаться взаимно заблокированным.
Q1		Очереди. Переполнение очереди задач. Возможна потеря информации или изменение поведения планировщика. Это может привести к пропуску предельных сроков расписания и к неправильной последовательности задач.
Q2		Очереди. Переполнение очереди сообщений. Сообщения могут быть пропущены, потеряны или задержаны, если очередь имеет неправильный размер либо сообщения не извлекаются своевременно, если только от этого нет специальной защиты.
Q3		Очереди. Переполнение рабочей очереди ядра. Рабочая очередь используется для постановки в очередь отложенной работы ядра, когда ядро уже занято другой заявкой и очередь заполнена. Отложенная работа ядра должна поступать в рабочую очередь из процедуры обслуживания прерывания. Рабочая очередь может переполниться, если частота прерываний слишком высока и ядро не успевает обработать задачи в отведенное время.
I1		Прерывания и исключения. Прерывания во время атомарных операций. Некоторые операции над глобальными данными должны завершиться до того, как последующие операции сможет вызвать другая задача. Прерывание, пришедшее в этот период, может нарушить операции, которые модифицируют или используют частично модифицированную структуру, либо прерывание может быть потеряно, если оно замаскировано во время критического участка кода.
I2		Прерывания и исключения. Нет обработчика прерывания. Для прерывания не определен обработчик. Если пользователь не задал его, операционная система реального времени должна предоставить обработчик прерывания по умолчанию.
I3		Прерывания и исключения. Нет обработчика исключения. Для исключения, вызванного задачей, не определен обработчик. Должен существовать обработчик исключения по умолчанию, который приостанавливает задачу и сохраняет ее состояние в момент исключения.
I4		Прерывания и исключения. Сигнал возникает без соответствующего обработчика. Сигнал может быть послан задачей другой задаче или аппаратурой при определенных условиях исключения.
I5		Прерывания и исключения. Недостаточная защита супервизорной задачи. Супервизорная задача, вызванная из-за исключения, выполняется в незащищенном адресном пространстве, которое может быть повреждено.

Источник: V. Halwan и J. Krodel, *Study of Commercial Off-the-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications*, DOT/FAA/AR-02/118, Office of Aviation Research, Washington, DC, December 2002.

Литература

1. V. Halwan and J. Krodel, *Study of Commercial Off-the-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications*, DOT/FAA/AR-02/118 (Washington, DC: Office of Aviation Research, December 2002).

Приложение С. Вопросы, которые следует рассмотреть при выборе операционной системы реального времени (real-time operating system, RTOS; ОСРВ) для системы, критичной по безопасности

В главе 20 рассматриваются типичные характеристики, возможности и проблемы, связанные с ОСРВ. В этом приложении приведен перечень вопросов, которые следует учитывать при выборе ОСРВ для включения в систему, критичную по безопасности (как и в других частях этой книги, предполагается, что требуется соответствие DO-178B (КТ-178В) или DO-178C (КТ-178С)). Вопросы разделены на три категории: общие вопросы, вопросы по функциональности ОСРВ и вопросы по интеграции ОСРВ. Однако эти три категории взаимосвязаны. Помимо опыта автора, при составлении этого перечня вопросов использовались два источника: (1) пособие Федерального управления гражданской авиации США (Federal Aviation Administration, FAA) *Conducting software reviews prior to certification* [1] и (2) исследовательский отчет этого управления *Commercial off-the-shelf real-time operating system and architectural considerations* [2].

С.1 Общие вопросы

- Нужна ли вообще ОСРВ?
- Соответствует ли ОСРВ целям DO-178C (или DO-178B), и доступны ли подтверждающие данные ЖЦ ПО, показывающие соответствие? Будут ли предоставлены необходимые данные? Если нет, будут ли данные доступны для поддержки сертификации и поддержания летной годности? В консультативном циркуляре (Advisory Circular, AC) 20-148 перечислены типовые данные, предоставляемые вместе с сертифицируемым компонентом.
- Насколько сложна ОСРВ?
- Достаточно ли данных для поддержки требуемого уровня критичности? Если ОСРВ была получена методом обратной разработки, удовлетворяют ли процессы целям DO-178C (см. главу 25)?
- Были ли рассмотрены технические и сертификационные вопросы, касающиеся ОСРВ, указанные в главе 20, а также любые другие специфические для проекта вопросы?
- Реализованы ли типичные характеристики и возможности ОСРВ, перечисленные в главе 20? Если нет, влияет ли отсутствие той или иной характеристики или возможности на безопасность или соответствие требованиям?
- Какое влияние ОСРВ оказывает на жизненный цикл разработки системы, которая будет ее использовать?
- Совместима ли ОСРВ с другими разрабатываемыми компонентами? Достаточно ли она

гибка? Или она накладывает ограничения на разрабатываемую систему? Если ограничения есть, были ли они учтены? На каком языке и каким компилятором была разработана ОСРВ? Использует ли компилятор особенности процессора, такие как внеочередное выполнение, кэш и конвейеры? Является ли компилятор детерминированным? Какую инструментальную поддержку предоставляет ОСРВ?

- Как возможное устаревание аппаратуры повлияет на использование ОСРВ?
- Будет ли поставщик ОСРВ поддерживать требования поддержания летной годности для ОСРВ (например, поддержку на протяжении всего срока службы воздушного судна, которое ее использует)? Что произойдет, если поставщик уйдет с рынка? Обычно для таких ситуаций требуется какой-либо контракт или юридическое соглашение.
- Какая система регистрации сообщений о проблемах предусмотрена для этого продукта для поддержания летной годности?
- Какие есть гарантии того, что этот продукт не будет изменен без ведома пользователя? Если он будет модифицирован, какой процесс будет использоваться и как пользователь будет об этом уведомлен?
- Подготовлено ли точное руководство пользователя? Учитывает ли оно потребности разных пользователей? Или оно адаптировано под каждого пользователя отдельно? Какой интерфейс прикладного программирования (application program interface, API) используется? Поддерживает ли этот интерфейс переносимость? Обеспечивает ли ОСРВ всестороннюю поддержку этого интерфейса?
- Какие процессы управления конфигурацией и гарантии качества ПО использовались при разработке ОСРВ? Достаточно ли эти процессы для системы, использующей ее? Совместимы ли эти процессы с процессами управления конфигурацией и гарантии качества ПО системы, использующей ее?

С.2 Вопросы по функциональности ОСРВ

- Масштабируется ли ОСРВ при добавлении задач?
- Ограничено ли число задач во время выполнения? Если да, достаточен ли верхний предел для потребностей пользователя?
- Какие типы межзадачного взаимодействия поддерживаются? Можно ли настраивать квант времени?
- Предотвращает ли ОСРВ инверсию приоритетов?
- Поддерживает ли ОСРВ выбранный механизм синхронизации? Позволяет ли она пользователю выбирать алгоритм планирования? Поддерживают ли доступные подходы к

планированию требования безопасности? Приемлемо ли время переключения задач?

- Как ОСРВ обрабатывает следующие функции: (1) обработка задач, (2) управление памятью и (3) прерывания? Делает ли она это детерминированным образом?
- Как ОСРВ использует память, например внутренний и внешний кэш?
- Были ли рассмотрены следующие вопросы согласованности данных: повреждение или потеря данных, ошибочные результаты и аномальные параметры? Были ли рассмотрены типовые проблемы многозадачности, включая непреднамеренное завершение или удаление, переполнение памяти ядра и превышение размера стека?
- Были ли рассмотрены типовые проблемы планирования, включая повреждение блоков управления задачами, чрезмерную блокировку задач из-за инверсии приоритетов, взаимную блокировку, порождение задач, истощающих ресурсы центрального процессора, повреждение назначения приоритетов задач, сервисные вызовы с неограниченным временем выполнения и состояния гонки?
- Были ли рассмотрены проблемы памяти и ввода-вывода, такие как фрагментация кучи, неправильная адресация через указатели, перезапись данных, нарушение когерентности кэша, блокировка памяти, несанкционированный доступ к устройствам и неотслеживаемые ресурсы?
- Были ли в ОСРВ рассмотрены типовые проблемы очередей, включая переполнение очереди задач, переполнение очереди сообщений и переполнение рабочей очереди ядра?
- Были ли в ОСРВ рассмотрены вопросы, связанные с прерываниями и исключениями, например прерывания во время атомарных операций, отсутствие обработчика прерываний, отсутствие обработчика исключений и ненадлежащая защита супервизорной задачи? Если эта система используется для поддержки обособления и защиты (partitioning/protection), были ли рассмотрены вопросы обособления и защиты памяти, ввода-вывода и времени (см. главу 21)?
- Был ли в ОСРВ рассмотрен посторонний, мертвый или отключенный код? В частности, как неиспользуемые части этой системы учитываются в требованиях?
- Есть ли у ОСРВ или системы монитор состояния? Соответствуют ли механизмы восстановления после проблемных ситуаций потребностям безопасности системы?
- Как был рассмотрен анализ межкомпонентной связности по данным и по управлению? Соответствует ли он цели 8 таблицы A-7 DO-178C?
- Абстрагировано ли ядро от аппаратуры для поддержки переносимости на другие целевые вычислители?

- Если ОСРВ является готовым коммерческим ПО (commercial off-the-shelf software, COTS), были ли рассмотрены типовые проблемы готового коммерческого ПО (см. главу 24)?

С.3 Вопросы по интеграции ОСРВ

- Были ли выявлены потенциальные уязвимости ОСРВ? Является ли устранение тех уязвимостей, которые не устраняются проектными решениями ОСРВ, простым и выполнимым для интегратора или пользователя?
- Имеется ли паспорт изделия или спецификация, в которых сведены характеристики ОСРВ, ограничения, доступные сертификационные данные и так далее? Документирован ли подход к расчету характеристик и ограничений, указанных в этом документе?
- Определены ли временные характеристики ОСРВ, такие как задержки, джиттер переключения потоков, схемы планирования и уровни приоритета?
- Кто будет разрабатывать пакет поддержки платы (board support package, BSP) и драйверы устройств? Есть ли у команды разработки нужная информация для выполнения этой задачи?
- Доступны ли комплекты разработки, помогающие интеграции ОСРВ? Многие поставщики предоставляют комплекты, помогающие адаптировать ее, разрабатывать драйверы, разрабатывать пакеты поддержки платы и так далее.
- Легко ли конфигурируется ОСРВ для конкретного применения, или для этого требуется значительная доработка?
- Разрабатывалась ли ОСРВ для использования с конкретным процессором или семейством процессоров?
- Хорошо ли понятны интерфейсы между ОСРВ и центральным процессором (central processing unit, CPU)?
- Эффективно ли рассчитывается влияние ОСРВ на наихудшее время выполнения?
- Доступен ли исходный код для анализа разработчиками приложений более высокого уровня критичности, если это необходимо для поддержки низкоуровневой функциональности? На какие ресурсы аппаратуры влияет ОСРВ?
- Какое влияние на возможность сертификации системы оказывает ПО, используемое для изоляции ОСРВ от целевого вычислителя (например, пакет поддержки платы или драйверы устройств)?
- Какие инструменты доступны для анализа производительности ОСРВ, и могут ли эти инструменты верифицировать детерминированное поведение?

- Доступны ли инструменты для удаленной диагностики? Многие поставщики ОСРВ также предоставляют инструменты, которые могут помочь анализировать поведение системы и поддерживать работы по анализу и тестированию.
- Имеются ли у ОСРВ возможности информационной безопасности (security), конфликтующие с целями безопасности системы (safety)? Если да, как эти конфликты будут устранены? Есть ли у центрального процессора внутренний блок управления памятью (memory management unit, MMU), и поддерживает ли он обособление (partitioning)?
- Как обрабатывается межкомпонентная связность по данным и по управлению между ОСРВ и приложениями?

Литература

1. Федеральное управление гражданской авиации США, *Conducting Software Reviews Prior to Certification*, Aircraft Certification Service (Rev. 1, January 2004).
2. J. Krodel, *Commercial off-the-shelf real-time operating system and architectural considerations*, DOT/FAA/AR-03/77 (Washington, DC: Office of Aviation Research, February 2004).

Приложение D. Вопросы по истории эксплуатации ПО

В главе 24 были приведены четыре категории вопросов, которые следует учитывать при оценке истории эксплуатации ПО. Это приложение включает конкретные вопросы из *Руководства Федерального управления гражданской авиации США по истории эксплуатации ПО* [1]. Руководство доступно на сайте Федерального управления гражданской авиации США (www.faa.gov); однако эти вопросы приведены здесь для удобства.

D.1 Вопросы, касающиеся сообщений о проблемах [1]

1. Отслеживаются ли версии программного обеспечения на протяжении периода истории эксплуатации?
2. Отслеживаются ли сообщения о проблемах применительно к конкретным версиям ПО?
3. Связаны ли сообщения о проблемах с решениями/исправлениями и анализом влияния изменений?
4. Ведется ли история редакций/изменений для различных версий программного обеспечения?
5. Выполнялся ли анализ влияния изменений для внесенных изменений?
6. Сообщалось ли о проблемах, возникших в эксплуатации?
7. Были ли зарегистрированы все заявленные проблемы?
8. Хранились ли эти сообщения о проблемах в репозитории, из которого их можно извлечь?
9. Были ли эксплуатационные проблемы тщательно проанализированы и/или были ли эти анализы включены в сообщения о проблемах либо на них были даны надлежащие ссылки?
10. Классифицируются ли проблемы в репозитории сообщений о проблемах?
11. Если один и тот же тип проблемы сообщался многократно, создавались ли множественные записи или одна запись для конкретной проблемы?
12. Если в лаборатории при выполнении копий эксплуатационных версий ПО в течение периода истории эксплуатации обнаруживались проблемы, включались ли эти проблемы в систему сообщений о проблемах?
13. Отслеживается ли по каждому сообщению о проблеме его статус, то есть исправлена проблема или остается открытой?
14. Если проблема была исправлена, имеется ли запись о том, как именно она была исправлена: в требованиях, проекте, коде?

15. Есть ли запись о новой версии программного обеспечения с новым выпуском после исправления проблемы?
16. Есть ли проблемы, для которых отсутствует соответствующая запись об изменении версии программного обеспечения?
17. Показывает ли история изменений, что программное обеспечение в настоящее время стабильно и зрелое?
18. Обладает ли продукт свойством выдавать сообщение пользователю при возникновении ошибки? (Некоторые продукты могут не иметь средств перехвата ошибок, поэтому они могут просто продолжать выполнение с неверными результатами и без какого-либо указания на отказ.)
19. Дали ли поставщик или организация, собирающая сообщения о проблемах, всем пользователям ясно понять, что проблемы собираются и исправляются?
20. Классифицированы ли все проблемы в репозитории сообщений о проблемах?
21. Идентифицируются ли проблемы, связанные с безопасностью, как таковые? Можно ли извлечь проблемы, связанные с безопасностью?
22. Есть ли запись о том, какие связанные с безопасностью проблемы исправлены, а какие оставлены открытыми?
23. Достаточно ли данных после последнего исправления проблем, связанных с безопасностью, чтобы оценить, что проблема была исправлена и новых проблем, связанных с безопасностью, не возникло?
24. Оказывают ли открытые сообщения о проблемах какое-либо влияние на безопасность?
25. Достаточно ли данных после последнего исправления проблем, связанных с безопасностью, чтобы оценить, что проблема решена и новых проблем, связанных с безопасностью, не возникло?
26. Классифицируются ли сообщения о проблемах и их решения так, чтобы было видно, каким образом было реализовано исправление?
27. Можно ли проследить конкретные исправления по версиям выпуска и сделать вывод на основе исправлений в проекте и коде, что новые версии соответствуют этим исправлениям?
28. Можно ли отделить те сообщения о проблемах, которые были устранены в аппаратуре или изменением требований?
29. Связаны ли сообщения о проблемах с решениями/исправлениями и анализом измене-

ний?

30. Если решения указывали на изменение аппаратуры, режима использования или требований, имеется ли анализ того, не делают ли эти изменения недействительными данные истории эксплуатации, относящиеся к периоду до такого изменения?
31. Имеется ли исправление проблемы с изменениями в программном обеспечении, но без записи об изменении версии ПО?
32. Является ли определенный период эксплуатации подходящим с учетом природы рассматриваемого программного обеспечения?
33. Сколько копий программного обеспечения используется и отслеживается на предмет проблем?
34. Сколько из этих применений можно считать сходными по эксплуатации и среде?
35. Совпадают ли области входных и выходных данных между периодом эксплуатации и предполагаемым применением?
36. Если области входных и выходных данных различаются, можно ли скорректировать их с использованием связующего программного кода (glue code)?
37. Включает ли период эксплуатации нормальные и аномальные условия работы?
38. Имеется ли запись об общем числе обращений в службу поддержки, полученных за этот период?
39. Были ли предупреждения и прерывания обслуживания частью этой системы сообщений о проблемах?
40. Анализировались ли предупреждения, чтобы убедиться, являлись они проблемами или нет?
41. Использовалась ли процедура регистрации сообщений о проблемах как ошибок?
42. Какова была логика, лежащая в основе содержания этой процедуры?
43. Имеются ли доказательства того, что эта процедура обеспечивалась и применялась последовательно на протяжении всего периода истории эксплуатации?
44. Соответствует ли история гарантийных претензий по продукту тому роду проблем, которые наблюдаются в истории эксплуатации?
45. Были ли сообщения о проблемах, определенные в исходной области применения как не связанные с безопасностью, рассмотрены с целью установить, не связаны ли они с безопасностью в целевой области применения?

D.2 Вопросы, касающиеся эксплуатации [1]

1. Является ли предполагаемая эксплуатация программного обеспечения сходной с использованием в период истории эксплуатации, то есть по его интерфейсам с внешним миром, людьми и процедурами?
2. Были ли проанализированы различия между использованием в истории эксплуатации и предполагаемым использованием?
3. Есть ли различия в режимах работы при новом использовании?
4. Используются ли в эксплуатационном применении только некоторые функции предполагаемого приложения?
5. Выполнен ли анализ пробелов по функциям, которые нужны в предполагаемом приложении, но не использовались в период эксплуатации?
6. Является ли определение нормальной эксплуатации и времени нормальной эксплуатации подходящим для данного продукта?
7. Включает ли период эксплуатации нормальные и аномальные условия работы?
8. Имеется ли технологическое различие между использованием продукта в период истории эксплуатации и новым использованием, например ручной режим по сравнению с автоматическим, перехват ошибок пользователем, использование в сети по сравнению с автономным режимом и так далее?
9. Требовалось ли обучение операторов процедурам при использовании продукта в течение зафиксированного периода истории эксплуатации?
10. Имеется ли план предоставить аналогичное обучение при новой эксплуатации?
11. Будет ли уровень программного обеспечения для новой системы таким же, как и в старой системе?

D.3 Вопросы, касающиеся среды [1]

1. Являются ли аппаратная среда истории эксплуатации и целевая среда сходными?
2. Были ли проанализированы различия в ресурсах между двумя вычислительными системами, например время, память, точность, прецизионность, службы связи, встроенные тесты, устойчивость к неисправностям, каналы и порты, режимы очередей, приоритеты, действия по восстановлению после ошибок и так далее?
3. Одинаковы ли требования безопасности, с которыми продукт сталкивается в обеих средах?

4. Одинаковы ли исключительные ситуации, с которыми продукт сталкивается в обеих средах?
5. Доступны ли данные, необходимые для анализа сходства среды? (Такие данные обычно не входят в состав данных о проблемах.)
6. Показывает ли анализ, какие части данных истории эксплуатации применимы к предполагаемому использованию?
7. Какой объем зачета по истории эксплуатации можно отнести к самому продукту, в отличие от свойств устойчивости к неисправностям вычислительной среды в период истории эксплуатации?
8. Совместим ли продукт с целевым вычислителем без внесения изменений в программное обеспечение продукта?
9. Если аппаратные среды различаются, были ли эти различия проанализированы?
10. Имели ли место аппаратные модификации в течение периода истории эксплуатации?
11. Если аппаратные модификации имели место, остается ли уместным учитывать период истории эксплуатации до этих модификаций?
12. Нужны ли данные по требованиям и проекту программного обеспечения для анализа того, приемлемо ли управление конфигурацией всех изменений аппаратуры, отмеченных в истории эксплуатации?

D.4 Вопросы, касающиеся времени [1]

1. Как определяется период эксплуатации?
2. Является ли определенный период эксплуатации подходящим с учетом природы рассматриваемого программного обеспечения?
3. Как определяется время нормальной эксплуатации?
4. Включает ли время нормальной эксплуатации, используемое в периоде эксплуатации, нормальные и аномальные условия работы?
5. Можно ли вывести непрерывное время работы из данных истории эксплуатации?
6. Выделяется ли из всего доступного массива данных истории эксплуатации часть, относящаяся к релевантной эксплуатации?
7. Каков был критерий оценки длительности периода эксплуатации?
8. Сколько копий программного обеспечения используется и отслеживается на предмет проблем?

9. Какова продолжительность релевантной эксплуатации?
10. Является ли определение релевантной эксплуатации подходящим?
11. Используется ли именно эта продолжительность для расчета частоты ошибок?
12. Насколько надежным был способ измерения времени?
13. Насколько последовательным был способ измерения времени на протяжении всего периода истории эксплуатации?
14. Имеется ли у вас предложенная приемлемая частота ошибок, обоснованная и подходящая для уровня безопасности предполагаемого использования, еще до анализа данных истории эксплуатации?
15. Каким образом, по вашему предложению, должна вычисляться эта частота ошибок до анализа данных истории эксплуатации?
16. Является ли вычисление частоты ошибок, то есть общее число ошибок, деленное на длительность времени, на число циклов выполнения, на число событий, таких как посадки, на летные часы, на пройденную дистанцию полета или на суммарное время работы всей совокупности экземпляров, подходящим для рассматриваемого приложения? Какова была общая длительность времени, использованная для этого расчета? Было ли уделено внимание тому, чтобы учитывать только подходящие интервалы времени?
17. Какова фактическая частота ошибок, вычисленная после анализа данных истории эксплуатации?
18. Превышает ли эта частота ошибок предложенную приемлемую частоту ошибок, определенную в Плане сертификации ПО (Plan for Software Aspects of Certification, PSAC)?*
19. Если частота ошибок выше, проводился ли анализ для повторной оценки частоты ошибок?

Литература

1. U.D. Ferrell and T.K. Ferrell, *Software Service History Handbook*, DOT/FAA/AR-01/116 (Washington, DC: Office of Aviation Research, January 2002).